Urban Planning Processes

A Research Note[1]

Version 12. Incomplete Draft

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Abstract. [2] We examine concepts of urban planning. That is, we emphasize, in this research note, the processes of urban planning.[3] In so doing we abstract from the information (the 'data') that urban planning is based on and results in. We distinguish between two kinds of urban planning processes: the basic, 'ab initio' (or base), process of determining "the general layout of the land (!)", and the derived, 'follow-up', processes focused on social and technological infrastructures. Base urban planning applies to descriptions of "the land": geographic, that is, geodetic, geotechnical, meteorological, and, in general, such information as are based in the given nature. Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: *industry zones, office and shopping zones, residential zones, recreational areas, transport, electricity, water, waste, health care, schools, etc.* This incomplete research note also discusses issues of urban planning project management, cf. Sect. 5.5, and urban planning document, cf. Sect. 5.2.

Contents

1 Introduction
- 1.1 On Urban Planning 5
  - 1.1.1 Infrastructures 5
  - 1.1.2 Wikipedia: https://en.wikipedia.org/wiki/Urban_planning 5
  - 1.1.3 Theories of Urban Planning 6
    - Technical aspects 6
    - Urban planners 6
  - 1.1.4 References 6
- 1.2 A Triptych of Software Development 7
- 1.3 On Domain Modeling 7
- 1.4 On Formality 7
- 1.5 On Formal Notations 7
- 1.6 On the Form of This Research Note 8

2 An Urban Planning System

[1] © Dines Bjørner, 2017
Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK 2840 Holte, Denmark
[2] This is Version 12 of the present document. It was first issued 9 September 2017. Subsequent editions (10 September 2017: 05:36 am + 6 hours) of Version 12 will appear, from day to day, during September 2017.
[3] This research note is being prepared for my stay at the *China Intelligent Urbanization Co-creation Center for High Density Region*, College of Architecture & Urban Planning (CAUP, http://en.tongji-caup.org/) at TongJi University, Shanghai, in September 2017 as hosted by Prof. Otthein Herzog.

3 Base Urban Planning
3.1 Urban Planning Information Categories 9
3.1.1 "Input" 9
3.1.2 "Output" 9
3.2 The Iterative Nature of Urban Planning 10
3.3 Initialisation 10
3.3.1 Existing versus Evolving Plans 11
3.4 A Simple Functional Form 11
3.5 Oracles and Repositories 12
3.5.1 The Base 'Input' Oracle 12
3.5.2 The Base Resumption Repository 12
3.6 A Simple Behavioural Form 13
4 Derived Urban Plannings
4.1 Preliminaries 14
4.1.1 Derived Urban Plan Indices 14
4.1.2 A "Reservoir" of Derived Urban Planning Indices 14
4.1.3 A Derived Urban Planning Index Selector 15
4.1.4 The Derived Urban Plan Generator 15
4.1.5 The Revised Base Urban Planning Behaviour 15
4.2 The Derived Urban Planning Functions 16
4.3 The Derived Urban Planning Behaviour 16
4.4 The Derived Resumption Repository 17
4.4.1 The Consolidated Derived Resumption Map 17
4.4.2 The Consolidated Derived Resumption Repository Channel 17
4.4.3 The Consolidated Derived Resumption Repository 17
4.4.4 Initial Consolidated Derived Urban Plannings 17
4.4.5 Initialisation of The Derived Quintuplet Oracle 17
4.5 A Visual Rendition of Urban Planning Development 17
4.6 Revised Selection of Derived Urban Plannings 18
4.6.1 Review 18
4.6.2 A Potential Derived Urban Plan Indices Selector 19
4.6.3 A Revised Derived Urban Plan Index Set Selector 19
4.6.4 Revision of Derived Urban Plan Invocation 19
4.7 The Urban Planning System 19
5 Further Work
5.1 Reasoning About Deadlock, Starvation, Live-lock and Liveness 20
5.2 Document Handling 20
5.2.1 Information Categories 20
5.2.2 Urban Planning Documents 20
5.2.3 A Document Handling System 20
5.3 Validation and Verification (V&V) 21
5.4 Towards Formalising Urban Planning Information 21
Land Areas 21
Area Points and Geographic Data 22
Area Points and Social-Economic Data 22
5.4.1 Etcetera 22
5.5 Urban Planning Project Management 23
5.5.1 Urban Planning Projects 23
5.5.2 Strategic, Tactical and Operational Management 23
Project Resources 23
Strategic Management 24
Tactical Management 24
Operational Management 24
5.5.3 Urban Planning Management 24
6 Conclusion
7 Bibliography
7.1 Bibliographical Notes 24
7.2 Domain Modeling Experiments 24
7.3 References 25
A A Document System
A.1 Introduction 29
A.2 A Document Systems Description 29
A.3 A System for Managing, Archiving and Handling Documents 29
A.4 Principal Endurants 29
A.5 Unique Identifiers 30
A.6 Documents: A First View 30
A.6.1 Document Identifiers 30

A.6.2 Document Descriptors 30
A.6.3 Document Annotations 31
A.6.4 Document Contents: Text/Graphics 31
A.6.5 Document Histories 31
A.6.6 A Summary of Document Attributes 31
A.7 Behaviours: An Informal, First View 32
A.8 Channels, A First View 33
A.9 An Informal Graphical System Rendition 34
A.10 Behaviour Signatures 34
A.11 Time 35
A.11.1 Time and Time Intervals: Types and Functions 35
A.11.2 A Time Behaviour and a Time Channel 35
A.11.3 An Informal RSL Construct 35
A.12 Behaviour "States" 35
A.13 Inter-Behaviour Messages 36
A.13.1 Management Messages with Respect to the Archive 36
A.13.2 Management Messages with Respect to Handlers 37
A.13.3 Document Access Rights 37
A.13.4 Archive Messages with Respect to Management 37
A.13.5 Archive Message with Respect to Documents 38
A.13.6 Handler Messages with Respect to Documents 38
A.13.7 Handler Messages with Respect to Management 38
A.13.8 A Summary of Behaviour Interactions 38
A.14 A General Discussion of Handler and Document Interactions 38
A.15 Channels: A Final View 39
A.16 An Informal Summary of Behaviours 39
A.16.1 The Create Behaviour: Left Fig. 5 on Page 40 39
A.16.2 The Edit Behaviour: Right Fig. 5 on Page 40 40
A.16.3 The Read Behaviour: Left Fig. 6 on Page 40 40
A.16.4 The Copy Behaviour: Right Fig. 6 on Page 40 41
A.16.5 The Grant Behaviour: Left Fig. 7 on Page 41 41
A.16.6 The Shred Behaviour: Right Fig. 7 on Page 41 41
A.17 The Behaviour Actions 42
A.17.1 Management Behaviour 42
Management Create Behaviour: Left Fig. 5 on Page 40 42
Management Copy Behaviour: Right Fig. 6 on Page 40 43
Management Grant Behaviour: Left Fig. 7 on Page 41 44
Management Shred Behaviour: Right Fig. 7 on Page 41 44
A.17.2 Archive Behaviour 44
The Archive Create Behaviour: Left Fig. 5 on Page 40 45
The Archive Copy Behaviour: Right Fig. 6 on Page 40 45
The Archive Shred Behaviour: Right Fig. 7 on Page 41 46
A.17.3 Handler Behaviours 46
The Handler Create Behaviour: Left Fig. 5 on Page 40 46
The Handler Edit Behaviour: Right Fig. 5 on Page 40 46
The Handler Read Behaviour: Left Fig. 6 on Page 40 47
The Handler Copy Behaviour: Right Fig. 6 on Page 40 47
The Handler Grant Behaviour: Left Fig. 7 on Page 41 48
A.17.4 Document Behaviours 48
The Document Edit Behaviour: Right Fig. 5 on Page 40 48
The Document Read Behaviour: Left Fig. 6 on Page 40 48
The Document Shred Behaviour: Right Fig. 7 on Page 41 49
A.18 Conclusion 49
B RSL
C RSL: The RAISE Specification Language – A Primer
C.1 Type Expressions 50
C.1.1 Atomic Types 50
C.1.2 Composite Types 50
Concrete Composite Types 50
Sorts and Observer Functions 51
C.2 Type Definitions 51
C.2.1 Concrete Types 51
C.2.2 Subtypes 52
C.2.3 Sorts — Abstract Types 52
C.3 The RSL Predicate Calculus 52
C.4 Propositional Expressions 52
C.4.1 Simple Predicate Expressions 52

C.4.2 Quantified Expressions 53
C.5 Concrete RSL Types: Values and Operations 53
C.5.1 Arithmetic 53
C.5.2 Set Expressions 53
Set Enumerations 53
Set Comprehension 53
C.5.3 Cartesian Expressions 53
Cartesian Enumerations 53
C.5.4 List Expressions 54
List Enumerations 54
List Comprehension 54
C.5.5 Map Expressions 54
Map Enumerations 54
Map Comprehension 54
C.5.6 Set Operations 54
Set Operator Signatures 54
Set Examples 55
Informal Explication 55
Set Operator Definitions 55
C.5.7 Cartesian Operations 56
C.5.8 List Operations 56
List Operator Signatures 56
List Operation Examples 56
Informal Explication 56
List Operator Definitions 57
C.5.9 Map Operations 57
Map Operator Signatures and Map Operation Examples 57
Map Operation Explication 58
Map Operation Redefinitions 58
C.6 λ-Calculus + Functions 59
C.6.1 The λ-Calculus Syntax 59
C.6.2 Free and Bound Variables 59
C.6.3 Substitution 59
C.6.4 α-Renaming and β-Reduction 59
C.6.5 Function Signatures 59
C.6.6 Function Definitions 60
C.7 Other Applicative Expressions 60
C.7.1 Simple let Expressions 60
C.7.2 Recursive let Expressions 60
C.7.3 Predicative let Expressions 61
C.7.4 Pattern and "Wild Card" let Expressions 61
C.7.5 Conditionals 61
C.7.6 Operator/Operand Expressions 61
C.8 Imperative Constructs 62
C.8.1 Statements and State Changes 62
C.8.2 Variables and Assignment 62
C.8.3 Statement Sequences and skip 62
C.8.4 Imperative Conditionals 62
C.8.5 Iterative Conditionals 62
C.8.6 Iterative Sequencing 63
C.9 Process Constructs 63
C.9.1 Process Channels 63
C.9.2 Process Composition 63
C.9.3 Input/Output Events 63
C.9.4 Process Definitions 63
C.10 Simple RSL Specifications 64

1. Introduction

Urban planning is a technical and political process concerned with the development and use of land, planning permission, protection and use of the environment, public welfare, and the design of the urban environment,

including air, water, and the infrastructure passing into and out of urban areas, such as transportation, communications, and distribution networks.[4]

In this research note we shall try to understand one of the aspects of the domain underlying urban planning, namely that of some possible urban planning (development) processes. We are trying to understand and describe a domain, not requirements for IT for that domain and certainly not the IT (incl. its software).

1.1. On Urban Planning

We search for answers to the question: *"What is Urban Planning?"*. First we identiffy "planning areas". Then we sketch element of a first domain model for Urban Planning.

Urban planning seems to be also be about infrastructure planning. So we examine these terms.

1.1.1. Infrastructures

The term 'infrastructure' has gained currency in the last 70 years.[5]. It is more frequently used in socio-economic than in scientific, let alone computing science, contexts.

According to the World Bank, 'infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users). We take a more technical view, and see infrastructures as concerned with supporting other systems or activities.

Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of data, people and/or materials. Hence issues of openness, timeliness, security, lack of corruption and resilience are often important.

Examples of infrastructures, or, more precisely, infrastructure components, are:

- transport systems (roads, railways, air traffic, canals/rivers/lake/ocean , etc.);
- water and sewage;
- telecommunications;
- postal service (physical letters, packages etc.);
- power: electricity, gas, oil, wind (generation, distribution); etc.
- the financial industry (banking, insurance, securities, clearing, etc.);
- documents (creation, editing, formatting, etc.);
- ministry of finance (taxation, budget, treasury, etc.);
- health care (private physicians, clinics, hospitals, pharmacies, etc.);
- education (kindergartens, pre-schools, primary schools, secondary schools, high schools, colleges, universities);
- manufacturing industry;
- etcetera.

1.1.2. Wikipedia: https://en.wikipedia.org/wiki/Urban_planning

Urban planning is a technical and political process concerned with the **development and use of land** planning permission, protection and use of the environment, **public wellfare**, and the design of the urban environment, including **air, water, and the infrastructure** passing into and out of urban areas, such as **transportation, communications, and distribution networks** [1].

Urban planning is also referred to as urban and regional planning, regional planning, town planning, city planning, rural planning or some combination in various areas worldwide. It takes many forms and it can share perspectives and practices with urban design [2].

Urban planning guides orderly development in urban, suburban and rural areas. Although predominantly concerned with the planning of settlements and communities, urban planning is also responsible for the **planning and development of water use and resources, rural and agricultural land, parks and conserving areas of natural environmental** significance. Practitioners of urban planning are concerned with research

[4] https://en.wikipedia.org/wiki/Urban_planning

[5] Winston Churchill is quoted to have said, in the House of Commons, in 1936: *... the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as 'infrastructure' ...*

and analysis, strategic thinking, architecture, urban design, public consultation, policy recommendations, implementation and management [3].

Urban planners work with the cognate fields of architecture, landscape architecture, civil engineering, and public administration to achieve strategic, policy and sustainability goals. Early urban planners were often members of these cognate fields. Today urban planning is a separate, independent professional discipline. The discipline is the broader category that includes different sub-fields such as land-use planning, zoning, economic development, environmental planning, and transportation planning [4].

1.1.3. Theories of Urban Planning

Planning theory is the body of scientific concepts, definitions, behavioral relationships, and assumptions that define the body of knowledge of urban planning. There are eight procedural theories of planning that remain the principal theories of planning procedure today: the rational-comprehensive approach, the incremental approach, the transactive approach, the communicative approach, the advocacy approach, the equity approach, the radical approach, and the humanist or phenomenological approach [5].

Technical aspects Technical aspects of urban planning involve the applying scientific, technical processes, considerations and features that are involved in planning for land use, urban design, natural resources, transportation, and infrastructure. Urban planning includes techniques such as: predicting population growth, zoning, geographic mapping and analysis, analyzing park space, surveying the water supply, identifying transportation patterns, recognizing food supply demands, allocating healthcare and social services, and analyzing the impact of land use.

Urban planners An urban planner is a professional who works in the field of urban planning for the purpose of optimizing the effectiveness of a community's land use and infrastructure. They formulate plans for the development and management of urban and suburban areas, typically analyzing land use compatibility as well as economic, environmental and social trends. In developing the plan for a community (whether commercial, residential, agricultural, natural or recreational), urban planners must also consider a wide array of issues such as sustainability, air pollution, traffic congestion, crime, land values, legislation and zoning codes.

The importance of the urban planner is increasing throughout the 21st century, as modern society begins to face issues of increased population growth, climate change and unsustainable development. An urban planner could be considered a green collar professional.[clarification needed]

1.1.4. References

1 "What is Urban Planning" (retrieved April 24, 2015)
https://mcgill.ca/urbanplanning/planning

> Modern urban planning emerged as a profession in the early decades of the 20th century, largely as a response to the appalling **sanitary, social, and economic** conditions of rapidly-growing industrial cities. Initially the disciplines of architecture and civil engineering provided the nucleus of concerned professionals. They were joined by **public health** specialists, economists, sociologists, lawyers, and geographers, as the complexities of managing cities came to be more fully understood. Contemporary urban and regional planning techniques for survey, analysis, design, and implementation developed from an interdisciplinary synthesis of these fields.
>
> Today, urban planning can be described as a technical and political process concerned with the welfare of people, control of the use of land, design of the urban environment including transportation and communication networks, and protection and enhancement of the natural environment.

2 Van Assche, K., Beunen, R., Duineveld, M., & de Jong, H. (2013). Co-evolutions of planning and design: Risks and benefits of design perspectives in planning systems. Planning Theory, 12(2), 177-198.

3 Taylor, Nigel (2007). Urban Planning Theory since 1945, London, Sage.

4 https://www.planning.org/aboutplanning/whatisplanning.htm: "What Is Planning?". www.planning.org. Retrieved 2015-09-28.

5 https://www.planetizen.com/node/73570/how-planners-use-planning-theory: How Planners Use Planning Theory. Andrew Whittmore of the University of North Carolina Department of Urban and Regional Planning identifies planning theory in everyday practice.

1.2. A Triptych of Software Development

Before hardware and software systems can be designed and coded we must have a reasonable grasp of "its" requirements; before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain. To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

By a domain description we understand a collection of pairs of narrative and commensurate formal texts, where each pair describes either aspects of an endurant (i.e., a data) entity or aspects of a perdurant (i.e., an action, event or behaviour) entity.

1.3. On Domain Modeling

This research note is part of a series of experiments in domain modeling [24, 36, 42, 30, 28, 31, 18, 11, 62, 7, 64, 63, 6, 50, 4] – see Sect. 7.2 on Page 24. The concept of domain modeling is explored in a series of papers and reports [40, 38, 41, 37, 39, 33, 35, 23, 26, 13, 22, 48, 12]. The purpose of the present experiment, besides its hopeful contribution to *urban planning research & development* at TongJi University (Shanghai), is to explore modeling principles, techniques and tools not yet identified in [40, 38].

1.4. On Formality

We consider software programs to be formal, i.e., mathematical, quantities — rather than of social/psychological interest. We wish to be able to reason about software, whether programs, or program specifications, or requirements prescriptions, or domain descriptions. Although we shall only try to understand some facets of the domain of urban planning we shall eventually let such an understanding, in the form of a precise, formal, mathematical, although non-deterministic, i.e., "multiple choice", description be the basis for subsequent requirements prescriptions for software support, and, again, eventually, "the real software itself", that is, tools, for urban planners. We do so, so that we can argue, eventually prove formally, that the software **is correct** with respect to the (i.e., its) formally prescribed requirements, and that the software **meets customer**, i.e., domain users' **expectations** – as expressed in the formal domain description.

1.5. On Formal Notations

To be able to *prove formal correctness* and *meeting customer expectations* we avail ourselves of some formal notation. In this research note we use the `RAISE` [56] Specification Language, `RSL`, [55]. Other formal notations, such as `Alloy` [58], `Event B` [1], `VDM-SL` [51, 52, 54] or `Z` [65] could be used We choose `RSL` since it, to our taste, nicely embodies Hoare's concept of *Communicating Sequential Processes*, `CSP` [57]. In general we refer to the following set of textbooks [8]:

also published by QingHua University Press:

See [14, 15, 16, 19, 20, 21].

1.6. On the Form of This Research Note

The present form of this research note, as of 10 September 2017: 05:36 am, is that of recording a development. The development is that of *trying to come to grips with what urban planning is*. We have made the decision, from an early start, that urban planning *"as a whole"* is a collection of one base and an evolving number of (initially zero) derived urban planning behaviours. Here we have made the choice to model the various behaviours of a complex of urban planning functions.

2. An Urban Planning System

We think of urban planning to be "dividable" into base urban planning, base_up_beh, and derived urban plannings, der_up_beh, where subindex i indicate that there may be several, i.e., $i \in \{d_1, d_2, ..., d_n\}$, such derived urban plannings. We think of base urban planning to "convert" physical (geographic, that is, geodetic, geotechncal, meteorological, etc.) information about the land area to be developed into a base plan, that is, cartographic, cadestral and other such information (zoning, etc.). And we think of derived urban planning to "convert" base plans into technological and/or societal plans. Technological and societal urban planning concerns are typical such as *industry zones, office and shopping zones, residential zones, recreational areas, transport, electricity, water, waste, health care, schools, etc.* Each urban planning *behaviour*, whether 'base' or 'derived', is seen as a *sequence* of the application of "the same" urban planning *function*, i.e., an urban planning *action*. Each urban planning action takes a number of information *argument*s and yield information *result*s. The base urban planning behaviour may **start** one or more derived urban planning behaviours, der_up_beh$_i$, at the end of "completion" of a base urban planning *action*. Let (indices) $\{d_1, d_2, ..., d_n\}$ identify a set of separate derived urban plannings, each concerned with a distinct, reasonably delineated technological and/or societal urban planning concern. During a base urban planning development the actions start any of these derived urban plannings once. Thus we think of urban planning as a system of a single base urban planning process (i.e., behaviour), base_up_beh, which "spawns" zero, one or more (but a definite number of) derived urban planning processes (i.e., behaviours), der_up_beh$_i$. A derived urban planning processes, der_up_beh$_i$, may themselves start other derived urban planning processes, der_up_beh$_j$, der_up_beh$_k$, ..., der_up_beh$_\ell$. Figure 1 on the facing page is intended to illustrate the following: At time $t0$ a base urban planning is started. At time $t1$ the base urban planning initiates a number of derived urban development, $D1, ..., Di$. At time $t2$ the base urban planning initiates the Dj derived urban planning. At time $t3$ the derived urban planning Di initiates two derived urban plannings, Dk and $D\ell$. At time $t4$ the base urban planning ends. And at time $t5$ all urban plannings have ended. Urban planning actions are provided with "input" in the form of either geographic, geodetic, geotechnical, meteorological, etc., information, b_geo:bGEO[6], or auxiliary information, b_aux:bAUX, or requirements information, b_req:bREQ. The auxiliary ("management") information is such as *time and date, name (etc.) of information provider, "trustworthiness" of information*, etc. The requirements information serves to direct, to inform, the urban planners towards *what kind of urban plan* is desired.

[6] The b_ value prefixes and the b type prefixes shall designate base urban planning entities.

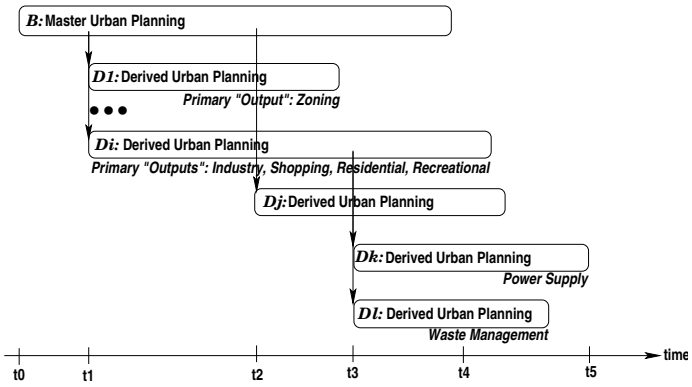

Fig. 1. An Urban Planning Development

3. Base Urban Planning

We begin this section with abstractions of the, perhaps, two most important aspects of urban planning, such as it may be seen by its individual practitioners: the information being handled: the "input", so-to-speak, to urban planning function(s) and these urban planning function(s). In two sections, in-between the information and the function sections (3.1 and 3.4), we very briefly discuss the iterative nature of urban planning, Sect. 3.2 on the next page, and initial values, Sect. 3.3, of the various information values.

3.1. Urban Planning Information Categories

3.1.1. "Input"

Among the arguments of urban planning are

6 information, bTSU, about *the urban space,* the demo-geographic area subject to planning: its geodetic "make-up", its geotechnical and meteorological properties, etc.[7]
7 related, but not geographic, information, bAUX[iliary][8],
8 and some requirements, bREQ[uirements].

type
6 bTSU
7 bAUX
8 bREQ

3.1.2. "Output"

Among results of urban planning are

9 "the plan" (or "plans"), bPLA[ns],
10 and possibly some other ancillary[9] documents, bANC[illerary].

type
9 bPLA
10 bANC

[7] We shall not include information about the use of the existing land. We refer to Slide 11 for how we handle such information.
[8] Auxiliary: giving help or support.
[9] Ancillary: providing necessary support to the main work

For this research note we shall leave the bTUS, bAUX, and bREQ argument types and the bPLA, and bANC result types further undefined[10] Typically bTSU would be be heavily text-annotated graphical documents which would show "the lay of the land", its geodetic[11], its geotechnical[12], and its meteorological[13], properties. Typically bAUX would then be some mixture of graphical and text documents that would "explain", in usually informal, casual ways, some of the relations between the geographic documents, when they were recorded, how they were vetted, their accuracy, etc. Typically bREQ would be informal, casual text documents, perhaps including pseudo-technical terms, which expresses expectations as to what the "powers-to-be" might consider suitable urban plans[14] and urban designs[15].

3.2. The Iterative Nature of Urban Planning

We take it that urban planning proceeds in "cycles":

11 In each cycle the base urban planning function, base_up_fct, is applied to an input argument triple, (b_tus,b_aux,b_req):(bTUS×bAUX×bREQ):bTRI, of "fresh" geodetic/geotechnical/meteorological (etc.), auxiliary and requirements information.

type
11 bTRI = bTUS × bAUX × bREQ

12 Each cycle, that is, each application of base_up_fct, results in a "most recent", not necessarily "final", plan and ancillary information, (b_pla,b_anc):bPLA×bANC:bRES.

type
12 bRES = bPLA × bANC

13 But, to "drive" the urban planning process, base_up_beh, towards "final", that is, an adequately satisfactory plan etc.,the urban planning function, base_up_fct, *need also be provided with the results of the previous iteration's result* — which we take to be a ("quintuplet") pair of an (i.e., the "previous") "input" triple and the previous result pair.

type
13 bQUI = bTRI × bRES

We shall refer to the input argument triple as 'the triplet' and the "driver" quintuplet (also) as a resumption. The above decisions on triplet arguments and quintuplet resumptions, including the latter's "feedback" to a next iteration function invocation is motivated as follows. We think of each invocation, i.e., step, of the urban planning function to "apply" itself to a small fragment of urban planning. Each such "small" step is to result in useful contributions to the evolving urban plan. The ancillary information emerging from each step informs about which aspects of urban planning was pursued in that step: where, in the plans, the outcome of those analysis and plan development can be seen. The reason for small step invocations are to allow ongoing reviews (not shown here), to pass on the intermediary results to other urban planning developments, etc. The decision to "feed" back "records" of the entire state of urban planning development motivated by the need for these "small step" invocations to analyse the ongoing, full state.

3.3. Initialisation

Urban planning proceeds in iterating from initial

[10] Understanding the bTUS, bAUX, bREQ, bPLA and the ANC types is a major urban planning issue.
[11] https://en.wikipedia.org/wiki/Geodesy
[12] https://en.wikipedia.org/wiki/Geotechnical_investigation
[13] https://en.wikipedia.org/wiki/Meteorology
[14] https://en.wikipedia.org/wiki/Urban_planning
[15] https://en.wikipedia.org/wiki/Urban_design

14 urban space, auxiliary and requirements information, as well as
15 (usually "empty") plans and ancillaries.

We extend the notion of initial values to

16 triplet arguments,
17 result pairs, and
18 "quintuplet" argument/result pairs.

towards such results (plans and ancillaries) that are deemed satisfactory.

```
value
14  b_tus_init: bTUS,
14  b_aux_init:bAUX,
14  b_req_init:bREQ
15 b_pla_init: bPLA,
15 b_anc_init:bANC
16 b_tri_init: bTRI = (b_tus_init,b_aux_init,b_req_init)
16           assert: b_tri_fit(b_tri_init,b_tri_init)
17 b_res_init: bRES = (b_pla_init,b_anc_init)
18 b_qui_init: bQUI = (b_tri_init,b_res_init)
```

We refer to Item 28 on the following page for an explanation of the b_tri_fit predicate.

3.3.1. Existing versus Evolving Plans

The quintuplet "feedback", which includes a 'plan' component, secures that possibly pre-existing plans are included, as initialised components of the plan results.[16] The iterative nature of urban planning thus allows for stepwise urban re-development, from existing "urban land-scapes" via mixed "previous" and "future" land-scapes to the final urban development plan.

3.4. A Simple Functional Form

19 The base urban planning *function*, base_up_fct, thus applies to

- (i) a "most recent" triplet of urban space, auxiliary and requirements information, and to
- (ii) a "past quintuplet", a resumption[17], that is, pair of urban space, auxiliary and requirements information as well as plan and ancillary information and yields such a resumption "quintuplet" pair of a triplet and a pair.

20 The application of base_up_fct to such arguments, i.e., base_up_fct(b_tus,b_aux,b_req)(b_qui) yields a "quintuplet" result, a resumption, b_qui:((b_tus′,b_aux′,b_req′),(b_pla,b_anc)).

We "explain" the relations between "input" arguments and "output" (**as**) results:

21 The "input" argument b_tri is "carried forward", b_tri′ (=b_tri), to be redeposited as part of the result.
22 The main part of the result, (b_pla,b_anc), is related, $\mathcal{P}$, to the input argument including the previous "result", the resumption.

```
19  base_up_fct: bTRI → bQUI → bQUI
20  base_up_fct(b_tri)(b_qui) as (b_tri′,(b_pla,b_anc))
21      b_tri = b_tri′ ∧
22      P_base(b_tri)(b_qui)(b_pla,b_anc)
```

For the time being we shall leave the base urban planning function, base_up_fct, that is, $\mathcal{P}_{\text{base}}$, uninterpreted.

[16] We refer to Item 6 on Page 9 and footnote 7 on Page 9.
[17] Resumption: like a repetition, a continuation

3.5. Oracles and Repositories

Oracles are simple behaviours that *offers* information to other behaviours. Repositories are simple behaviours that *store* information from other behaviours and *offers* stored information to other behaviours.

3.5.1. The Base 'Input' Oracle

An urban planning oracle, when so requested, will select some information – usually in some non-deterministic fashion, and usually subject to some constraint – and present this information to the requestor, i.e., an urban planning behaviour. In this section, i.e. Sect. 3.5.1, we shall deal with one specific oracle, b_tri_beh: one that "assembles" triplets, b_tri, of urban space, b_tus:bTUS, auxiliary, b_aux:bAUX, and requirements, b_req:bREQ, information to requesting behaviours. We introduce a pair of specification components:

23 a *channel*, b_tri_ch, over which a base urban planning behaviour, base_up_beh, offers to receive triplets, b_tri:bTRI, from an oracle, b_tri_beh,

24 and an *oracle*, b_tri_beh, which "remembers" its most recently communicated triplet[18].

```
channel
23  b_tri_ch:bTRI
value
24  b_tri_beh: bTRI → out b_tri_ch  Unit
```

25 The oracle assembles (b_tri′:bTRI), a base triplet which satisfies a predicate b_tri_fit(b_tri,b_tri′) – see Item 28.

26 That triplet is offered, b_tri_ch ! b_tri′, to the base urban behaviour –

27 whereupon the oracle resumes being the oracle, now, however, with the recently assembled base triplet as its resumption.

```
24  b_tri_beh(b_tri) ≡
25      let b_tri′:bTRI • b_tri_fit(b_tri,b_tri′) in
26      b_tri_ch ! b_tri′ ;
27      b_tri_beh(b_tri′)
24      end
29  pre: b_tri_fit(b_tri,b_tri)
```

28 The fitness predicate, b_tri_fit(b_tri,b_tri′), checks whether a "newly" assembled base triplet, b_tri′, stands in some suitable[19] relation $\mathcal{P}$(b_tri,b_tri′) to a a similar (f.ex., "earlier") base triplet, b_tri.

29 The fitness predicate holds for b_tri_fit(b_tri,b_tri).

```
28  b_tri_fit: bTRI × bTRI → Bool
28  b_tri_fit(tri,tri′) ≡ 𝒫(b_tri,b_tri′)
```

30 The oracle, b_tri_beh, is initialised with an initial triplet value b_tri_init, cf. formula Item 16 on the preceding page.

```
28  b_tri_beh(b_tri_init): assert: b_tri_fit(b_tri_init,b_tri_init)
```

3.5.2. The Base Resumption Repository

The "quintuplet" pair of an "input" triple and a result pair, b_qui:(b_tri:bTRI,(b_pla:bPLA,b_anc:bANC)) is thought of as residing in a *repository* behaviour, b_qui_beh, which "receives" (b_qui_ch?) "quintuplets" from the urban planning behaviour, or "offers" (b_qui_ch!b_qui) such to the urban planning behaviour.

[18] The oracle is initialised with b_tri_beh(geo_init,b_aux_init,b_req_init).
[19] – to be defined for each specific urban planning project

31 There is therefore a channel, b_qui_ch, between the urban planning behaviour and the "quintuplet" repository behaviour,

32 b_qui_beh.

33 It either

34 accepts or

35 offers quintuplets.

```
channel
31   b_qui_ch:bQUI
value
32   b_qui_beh: bQUI → in,out b_qui_ch  Unit
32   b_qui_beh(b_qui) ≡
34       b_qui_beh(b_qui_ch?)
33       ⌷
35       b_qui_ch!(b_qui) ; b_qui_beh(b_qui)
```

3.6. A Simple Behavioural Form

Urban planning, however, is a time-consuming "affair". So we model it as a behaviour.

36 The base_up_beh_0[20] behaviour takes no argument, hence the left signature element: **Unit**, avails itself of the input channel for obtaining proper input, b_tri, and b_qui, for the base urban function, base_up_fct, and output channel, for depositing a resumption, b_qui′, and (then) "goes on forever", as indicated by the right signature element: **Unit**.

```
36   base_up_beh_0: Unit → in b_tri_ch out b_qui_ch Unit
```

37 The simple (version of the) base_up_beh_0 behaviour

38 obtains the base triplet, b_tri, and the base resumption, b_qui, information,

39 performs the base_up_fct planning function and

40 provides its result, a resumption, b_qui′, to the base quintuplet repository,

41 whereupon it reverts to being base_up_beh_0.

```
value
37   base_up_beh_0() ≡
38       let (b_tri,b_qui) = (b_tri_ch?,b_qui_ch?) in
39       let b_qui′ = base_up_fct(b_tri)(b_qui) in
40       b_qui_ch ! b_qui′ end end ;
41       base_up_beh_0()
```

The base_up_beh_0 behaviour repeatedly "performs" urban planning, "from scratch", as if new urban space, auxiliary and requirements information was "new" in every re-planning — "ad infinitum"! We now revise base_up_beh_0 into base_up_beh_1 — a behaviour "almost" like base_up_beh_0, but one which may terminate.

42 base_up_beh_1

43 first behaves like base_up_beh_0 (Items 38–40)

44 then checks whether the obtained base resumption is satisfactory, that is, is OK as an end-result of base urban planning.

45 If so then base_up_beh_1 terminates,

46 else it resumes being base_up_beh_1.

[20] As there will be several versions, from simple towards more elaborate, of the base_up_beh behaviour, we index them.

```
value
36  base_up_beh_1: Unit → in b_tri_ch out b_qui_ch Unit
42  base_up_beh_1() ≡
43      let b_qui = b_base_up_fct(b_tri_ch?)(b_qui_ch?) in b_qui_ch!b_qui ;
44      if b_qui_satisfactory(b_qui)
45          then skip
46          else base_up_beh_1() end
42      end

44  b_qui_satisfactory: bQUI → Bool
```

The b_qui_satisfactory predicate inquires the base quintuplet, b_qui, as for its suitability as a final candidate for an urban plan[21].

4. Derived Urban Plannings

4.1. Preliminaries

4.1.1. Derived Urban Plan Indices

We think of *base* urban planning function, modeled by base_up_fct, as being concerned with the overall "division" of the urban space (i.e., geographical area, land and water) into zones for building, recreation, and other (i.e., the *master plan*). Aggregations of these zones, one, more or all (usually several), can then be further "[derive] planned" into

- (d_1) light, medium and heavy industry zones,
- (d_2) mixed shopping and residential zones,
- (d_3) apartment bldg. zones,
- ..., etc., etc.,
- (d_{m-1}) villa zones, and
- (d_m) recreational zones.

Additional forms of derived plannings are:

- (d_{m+1}) transport,
- (d_{m+2}) electricity supply,
- (d_{m+3}) water supply,
- (d_{m+4}) waste management,
- (d_{m+5}) health care,
- (d_{m+6}) fire brigades,
- ..., etc., etc.,
- (d_{m+n}) schools.

We refer to the d_i's as derived urban plan indices.

47 We think of this variety of "derived" plannings as indexed such as hinted at above,
48 and dups as the set of all indices.

```
type
47  DP == {|d1,d2,...,dn|}
value
48  dups:DP-set = {d1,d2,...,dn}
```

4.1.2. A "Reservoir" of Derived Urban Planning Indices

49 To secure that at most one derived planning, d_i (per i), is initiated we introduce a global variable, dps_var, initialised to an empty set of derived planning tokens and updated with the addition of selected DP tokens.

[21] The b_qui_satisfactory argument, b_qui, embodies not only that plan, but also the basis for its determination.

variable
49 dps_var:DP-**set** := {} **comment** dps_var *denotes a reference*

4.1.3. A Derived Urban Planning Index Selector

50 A function, sel_dps, selects zero, one or more "fresh" DP indices, that is, DP tokens that have not been selected before.

value
50 sel_dps: **Unit** → DP-**set**
50 sel_dps() ≡
50 **let** dps:DP-**set**•dps⊆dups \ **c** dps_var **in**
50 dps_var := **c** dps_var ∪ dps; dps **end**
comment
49 [**c** *denotes a contents-taking operator*]

We shall revise the above selector in Sect. 4.6.3 on Page 19.

4.1.4. The Derived Urban Plan Generator

51 We therefore edit the base_up_beh_1 behaviour slightly into the revised base_up_beh_2. In base_up_beh_2 we insert, "in parallel" (‖) with the "resumption" of base_up_beh_2 (cf. Item 46 on Page 13), an internal non-deterministic choice behaviour, der_up()[22]. It specifies the selection of zero, one of more DP tokens, and initiates corresponding derived planning behaviours, der_up_beh$_i$(), as well as their corresponding "input" triplet oracles, d_tri_beh$_i$(). But only at most once. These derived planning behaviours, der_up_beh$_i$, and "input" triplet oracles, d_tri_beh$_i$() are like base_up_beh_1, respectively b_tri_beh, only now they are "tuned" to the specific derived planning issues (i.e., $_i$).

When behaviour and function invocations where the names of these behaviors or functions names are prefixed with der_, e.g., der_name, and are indexed by some $_i$, i.e., der_name$_i$, then we mean the invocation of one specific i indexed behaviour or function from the indexed set of such, as defined by their behaviour and function definitions, see below.

value
51 der_up: **Unit** → **Unit**
51 der_up() ≡ **let** dps = sel_dps() **in** ‖{der_up_beh$_i$()‖d_tri_beh$_i$()|i:DP•i ∈ dps} **end**

We shall introduce the der_up_beh$_i$ and d_tri_beh$_i$ behaviours below.

4.1.5. The Revised Base Urban Planning Behaviour

We "take over" the basic structure and definition ("contents") of the urban planning function and behaviour from that of the base versions.

52 We think of zero, one or more derived plannings (der_up_beh$_1$, der_up_beh$_2$, ..., der_up_beh$_n$) being initiated after some stage of *base* function, base_up_fct, has concluded.

value
42" base_up_beh_2() ≡
46" **let** b_qui=base_up_fct(b_tri_ch?)(b_qui_ch?) **in** b_qui_ch!b_qui ;
44" **if** base_satisfactory(b_qui)
42" **then skip**
43" **else**
52 der_up() ‖
45" base_up_beh_2() **end end**

[22]

4.2. The Derived Urban Planning Functions

An important form of information for each derived urban planning function is the resumption, i.e., the quintuplet information from the base urban behaviour: bQui.

53 The new forms of information are: the derived urban planning auxiliary, dAUX_i, the derived urban planning requirements information, dREQ_i, as well as the derived urban planning plans, dPLA_i, and their ancillary information, dANC_i.

54 The primary arguments for the derived urban planning function, base_up_fct, is therefore a "quintuplet", d_qui:dQUI, of a base triplet, b_tri:bTRI, and the pair of the derived urban planning auxiliary information, $\mathsf{d\_aux}_i$:dAUX_i, and the derived urban planning requirements, $\mathsf{d\_req}_i$:dREQ_i.

The result of derived urban planning function, der_up_fct, as for the base urban planning function, base_up_fct,

55 is that of a "quintuplet", also a resumption, dQUI_i, of the primary arguments, b_tri:bTRI, and

56 the result, a pair of a derived plan, $\mathsf{d\_pla}_i$, and derived ancillaries, $\mathsf{d\_anc}_i$.

57 As for the base urban planning function, base_up_fct, it has a secondary, derived "quintuplet" argument (which, as for base_up_fct, helps "kick-start" urban planning). This second argument is the result of a previous application of the der_up_fct.

58 The derived urban planning function $\mathsf{der\_up\_fct}_i$ signature is therefore that of a function from a triplet of a most recent base quintuplet, derived urban planning auxiliary and derived urban planning requirements information to functions from derived "quintuplet" arguments to derived "quintuplet" results.

59 The triplet argument, $\mathsf{d\_tri}_i$, and the first part of the result, also a triplet, $\mathsf{d\_tri}'_i$, are the same.

60 The derived urban planning function $\mathsf{der\_up\_fct}_i$ is further characterised by a predicate, $\mathcal{P}_{\mathsf{der}_i}$, which we leave further undefined.

```
type
53  dAUX1, dAUX2, ..., dAUXn
53  dREQ1, dREQ2, ..., dREQn
53  dPLA1, dPLA2, ..., dPLAn
53  dANC1, dANC2, ..., dANCn
54  dTRIi = bQUI×dAUXi×dREQi      [i:DP•i∈dups]
56  dRESi = dPLAi × dANCi         [i:DP•i∈dups]
55  dQUIi = dTRIi×dRESi           [i:DP•i∈dups]
value
57  der_up_fcti: dTRIi → dQUImi → dQUIi  i:DP
58  der_up_fcti(d_trii)(d_quii) as (d_tri'i,d_resi)
59      d_trii = d_tri'i ∧
60      Pder_i(d_tri'i,d_resi)

60  Pder_i: dTRIi × dRESi → Bool
```

4.3. The Derived Urban Planning Behaviour

61 We think of zero, one or more derived plannings ($\mathsf{der\_up\_beh}_{i_1}$, $\mathsf{der\_up\_beh}_{i_2}$, ..., $\mathsf{der\_up\_beh}_{i_m}$) being initiated after some stage of the *der_up_fct*$_i$ function has concluded.

```
value
42"'  der_up_behi() ≡
46"'      let d_quii = der_up_fcti(d_tri_ch[i]?)(d_qui_ch[i]?) in
46"'      d_qui_ch[i] ! d_quii ;
44"'      if der_satisfactoryi(d_quii)
42"'          then skip
43"'          else
61                der_up() ‖
```

45"' der_up_beh$_i$() **end end**

4.4. The Derived Resumption Repository

4.4.1. The Consolidated Derived Resumption Map

62 The derived urban planning functions (and thus behaviours) operate, not on simple resumptions, as do the base urban planning functions (and behaviours), but on the aggregation of all derived functions' (etc.) quintuplets, that is, an indexed set of quintuplets – modeled as a derived resumptions map.

type
62 dQUIm = DP $\overrightarrow{m}$ dQUI$_i$

4.4.2. The Consolidated Derived Resumption Repository Channel

63 Communications between the individual derived urban planning behaviours and the consolidated derived resumption repository are via an indexed set of channels communicating derived resumptions maps.

channel
63 {d_qui_ch[i]:dQUIm|i:DP• i ∈ dups}

4.4.3. The Consolidated Derived Resumption Repository

64 The consolidated derived resumption repository behaviour either ([]) updates its state map with received individual derived resumptions, or offers the entire such state maps to whichever derived urban planning behaviour so requests.

value
64 d_qui_beh: dQUIm → **in**,**out** der_qui_ch[i] **Unit** i:DP
64 d_qui_beh(d_qui_m) ≡
64 (d_qui_beh(d_qui_m†[i↦d_qui_ch[i]?])
64 []
64 d_qui_ch[i]!(d_qui_m)) ;
64 d_qui_beh(d_qui_m)

4.4.4. Initial Consolidated Derived Urban Plannings

value
d_qui_m = [d$_1$ ↦ init_d_qui$_1$, ..., d$_n$ ↦ init_d_qui$_n$]

4.4.5. Initialisation of The Derived Quintuplet Oracle

As for base oracle and repository behaviours we initialise the derived quintuplet oracle:

der_qui_beh(init_d_qui_m)

4.5. A Visual Rendition of Urban Planning Development

The urban planning domain, when "operating at full speed", consists of the base urban planning behaviour (i.e., project), zero, one or more derived urban planning behaviours, each of the latter initiated by either the base urban planning project or a derived urban planning project. See Fig. 2 on the next page. The planning

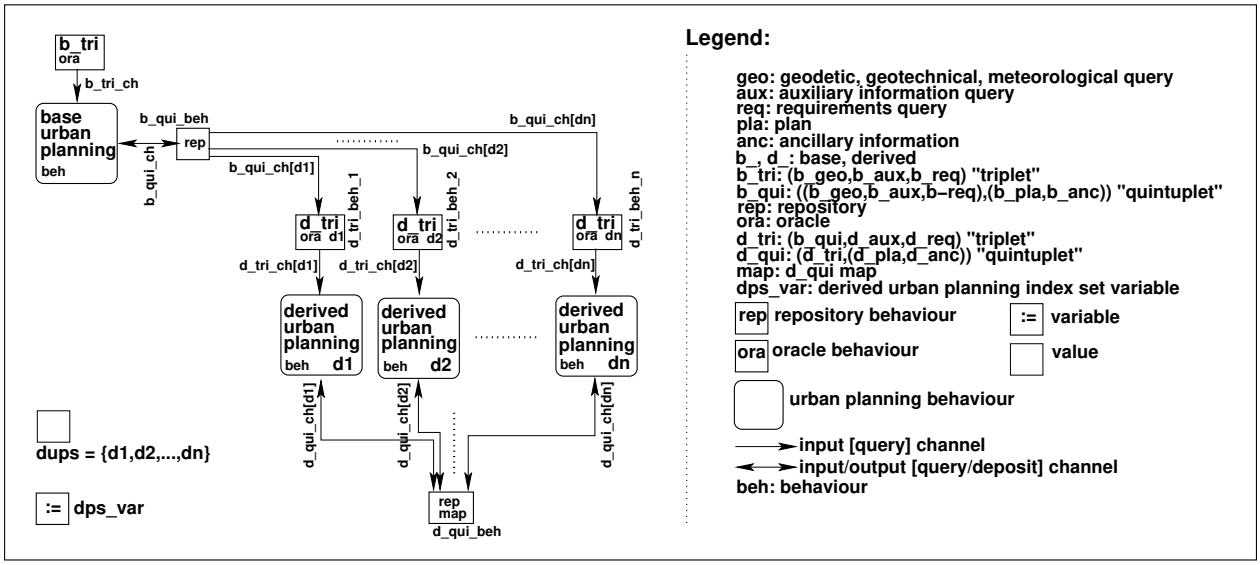

Fig. 2. An Urban Planning: n+1 Planning Behaviours, 2 Repository Behaviours, n+1 Oracles, a Variable, a Value and $3n$+2 Channels

behaviours, both the base and the derived, invoke respective urban planning functions, and these produce, such as we have modeled them, quintuplets of information, which are deposited with respective quintuplet repository behaviours: the base quintuplet repository behaviour, and the derived quintuplet repository behaviour — which maintains these quintuplets for all (invoked and thus ongoing) derived urban planning projects. We kindly ask you to review Fig. 2. All you have to 'master' is the fact that there is one base urban planning project, with its repository of base urban planning "quintuplets", and between 0 and n derived urban planning projects, with their shared, derived urban planning "quintuplets", Then there are the channels: the query (input) channels providing auxiliary and requirements information to both the one base urban planning project and the n derived urban planning projects; and the query/repository channels providing "quintuplet" aggregated information to the base urban planning project, as well as "quintuplet" aggregated information to the derived urban planning projects. Finally there are the "global" value representing the index set of derived urban planning indices, and variable which holds the index set of derived urban planning indices of ongoing derived urban planning projects.

4.6. Revised Selection of Derived Urban Plannings

4.6.1. Review

The derived urban planning generator function, der_up, cf. Item 51 on Page 15,

```
value
51  der_up: Unit → Unit
51  der_up() ≡
51    let dps = sel_dps() in
51    ‖ {der_up_beh_i()‖d_tri_beh_i()|i:DP•i ∈ dps} end
```

was invoked with no arguments, der_up(), cf. Item 52 on Page 15 and Item 61 on Page 16

```
52          der_up() ‖ [ respectively ]
61          der_up() ‖
```

4.6.2. A Potential Derived Urban Plan Indices Selector

Selection of potential derived urban planning indices was therefore rather arbitrary. We now let the selection depend on the aggregated resumption state of all (ongoing and) derived urban planning behaviours.

65 Function sel_dups examines either the base resumption or the aggregated resumption state of all (ongoing and) derived urban planning behaviours and yields a set of derived urban planning indices.

66 How it does that is, of course, not defined here.

```
value
65  sel_dups: (bQUI|dQUIm) → DP-set
66  sel_dups(dquim) ≡ ...
```

4.6.3. A Revised Derived Urban Plan Index Set Selector

67 We revise the derived urban plan index selector function give earlier, cf. Item 50 on Page 15. A function, sel_dps, selects zero, one or more DP "fresh" indices, that is, DP tokens that have not been selected before.

```
value
67  sel_dps: DP-set → DP-set
67  sel_dps(dups) ≡
67      let dps:DP-set•dps⊆dups∩dups \ c dps_var in
67      dps_var := c dps_var ∪ dps; dps end
```

4.6.4. Revision of Derived Urban Plan Invocation

We need to revise the two occurrences of der_up() – in the base urban planning behaviour, and in the (indexed set of) derived urban planning behaviours. Thus

```
52          der_up() ‖ [ respectively ]
61          der_up() ‖
```

is to be replaced by:

```
52          der_up(b_qui_ch?) ‖ ... [ respectively ]
61          der_up(d_qui_ch[ i ]?) ‖ ...
```

4.7. The Urban Planning System

68 Finally we can define an urban planning development as a system of concurrent behaviours:

- the base urban planning behaviour,
- the base "quintuplet" repository and
- the derived and consolidate "quintuplet" repository

```
value
68  up_sys: Unit → Unit
68  up_sys() ≡ base_up_beh() ‖ b_qui_beh(b_qui_init) ‖ d_qui_beh(d_qui_m)
```

Recall that the derived urban planning behaviours as well as the derived triplet behaviours are started by the base as well as the derived urban planning behaviours.

5. Further Work

5.1. Reasoning About Deadlock, Starvation, Live-lock and Liveness

The current author is quite unhappy about the way in which he has defined the urban planning, oracle and repository behaviours. Such issues as which invariants are maintained across behaviours are not addressed. In fact, it seems to be good practice, following Dijkstra, Lamport and others, to formulate appropriate such invariants and then "derive" behaviour definitions accordingly. In a rewrite of this research note, if ever, into a proper paper, the current author hopes to follow proper practices. He hopes to find younger talent to co-author this effort.

5.2. Document Handling

I may appear odd to the reader that I now turn to document handling. One central aspect of urban planning, strange, perhaps, to the reader, is that of handling the "zillions upon zillions" of documents that enter into and accrue from urban planning. If handling of these documents is not done properly a true nightmare will occur. So we shall briefly examine the urban planning document situation! From that we conclude that we must first try understand:

- What do we mean by a document?

5.2.1. Information Categories

Urban planning consumes information, most likely in the form of documents and produces urban planning documents, i.e., plans — referred to in Sects. 3–4 as information ("contained" in triplets and in quintuplets). We remind the reader of the types of these documents.

```
type
6   bGEO
7   bAUX
8   bREQ
9   bPLA
10  bANC
11  bTRI = bGEO × bAUX × bREQ
12  bRES = bPLA × bANC
13  bQUI = bTRI × bRES

53  dAUX1, dAUX2, ..., dAUXn
53  dREQ1, dREQ2, ..., dREQn
53  dPLA1, dPLA2, ..., dPLAn
53  dANC1, dANC2, ..., dANCn
54  dTRIi = bQUI×dAUXi×dREQi  [i:DP • i∈dups]
56  dRESi = dPLAi × dANCi     [i:DP • i∈dups]
55  dQUIi = dTRIi×dRESi       [i:DP • i∈dups]
62  dQUIm = DP →m dQUIi       [i:DP • i∈dups]
```

5.2.2. Urban Planning Documents

The urban planning functions and the urban planning behaviours, including both the base and the n derived variants, rely on documents. These documents are **created**, **edited**, **read**, **copied**, and, eventually, **shredded** by urban-planners. Editing documents result in new versions of "the same" document. While a document is being **edited** or **read** we think of it as not being **accessible** to other urban-planners. If urban-planners need to read a latest version of a document while that version is subject to editing by another urban planner, copies must first be made, before editing, one for each "needy" reader. Once, editing has and readings have finished, the "reader" copies need, or can, be shredded.

5.2.3. A Document Handling System

In Appendix A, [43], we sketch a document handling system domain. That is, not a document handling software system, not even requirements for a document handling software system, but just a description which, in essence, models documents and urban planners' actions on documents. (The urban planners are referred to as document handlers.) The description further models two 'aggregate' notions: one of 'handler management', and one of 'document archive'. Both seem necessary in order to "sort out" the granting of document access rights (that is, permissions to perform operations on documents), and the creation and shredding of documents, and in order to avoid dead-locks in access to and handling of documents.

5.3. Validation and Verification (V&V)

By **validation** of a document we shall mean: the primarily informal and social process of checking that the document description meets customer expectations.

Validation serves to get the right product.

By **verification** of a document we shall mean: the primarily formal, i.e., mathematical process of checking, testing and formal prooof that the model, which the document description entails, satisfies a number of properties.

Verification serves to get the product right.

By **validation of the urban planning model** of this document we shall understand the social process of explaining the model to urban planning stakehiolders, to obtain their reaction, and to possibly change the model according to stakeholder objections.

By **verification of the urban planning model** of this document we shall understand the formal process, based on formalisations (cf. Sect. 5.4) of the argument and result types of the description, of testing, model checking and formally proving properties of the model.

5.4. Towards Formalising Urban Planning Information

Land Areas The planning information to be further analysed with a view towards a partial formalisation and the statement of formal properties is: bGEO, bAUX, bREQ, bPLA, and bANC, cf. Items 6–10.

69 From the land (and sea) to be urban (re-)planned one can observe geographic, that is, geodetic, geotechnical and meteorological facts, GEO,

70 and social economic facts, SocEco.

71 From the geographic facts we can observe the individual geodetic, geotechnical and meteorological facts: GeoD, GeoT, Mete.

72 From social economic facts, SocEco we can observe individual social and individual economic facts, SOC, resp. ECO.

```
type
69  LAND, GEO, SocEco
value
69  obs_GEO: LAND → GEO
70  obs_SocEco: LAND → SocEco
type
71  GeoD, GeoT, Mete
value
71  obs_GeoD: GEO → GeoD
71  obs_GeoT: GEO → GeoT
71  obs_Mete: GEO → Mete
70  obs_Soc: SocEco → SOC
70  obs_Eco: SocEco → ECO
```

73 With land, geography, geographic, geotechnical and meteorological facts we can associate area:Area attributes,

74 and with social and economic facts we can also associate area:Area attributes

75 An containment predicate, is_in(a,a′), holds if area a is contained within area a′.

```
type
73  Area
value
73  attr_Area: (LAND|GEO|GeoD|GeoT|Mete) → Area
74  attr_Area: SOC|ECO → Area
75  is_in: Area × Area → Bool
```

76 For any land it is the case that for area attributes, a, of that land, that the area attribute, a', of the observed geography is contained within the land area, a, and that the area attributes, a'',

a of the area of the geodecy of that geography,
b of the area of the geotechnics of that geography, and
c of the area of the meteorology of that geography.

are contained within the geography area attribute.

axiom
76 ∀ l:LAND • is_in(attr_Area(obs_GEO(l)),attr_Area(l))
76a ∧ is_in(attr_Area(obs_GeoD(obs_GEO(l))),attr_Area(obs_GEO(l)))
76b ∧ is_in(attr_Area(obs_GeoT(obs_GEO(l))),attr_Area(obs_GEO(l)))
76c ∧ is_in(attr_Area(obs_Mete(obs_GEO(l))),attr_Area(obs_GEO(l)))

Etcetera for social and economic data.

Area Points and Geographic Data

77 We may think of a land area as consisting of a dense, in[de]finite set of (land area) points, p:P.
78 A land area point is either within an area (on the edge (or border or perimeter) of the area) or properly within the area[23].

type
77 Pt
value
78 is_within: (Pt|Area) × GEO → **Bool**

79 With each point of an area of geography we can associate a(ny) number of geodetic, geotechnical and meteorological data: their attribute types and values. Let A_i name the type of an indefinite number (n) of geodetic, geotechnical, meteorological, social, economic, and other more-or-less quantifiable attributes.
80 For a given point or area in or within the area of a given geography, pta, one can "extract" a summary, summary:Attr_Sum, for a set, atrs, of one or more attributes as observed either at a given time, t:T, **or** over a given interval, ti:TI, of time after a given time, discretised artgiven interval points, textsfti′, **or** summarised ("averaged") over the interval between to gievn times, t,t′. We do not define what we mean by an attribute summary.

type
79 A = $A_1|A_2|...|A_i|...|A_n$
value
79 attr_A_i: (Pt|Area)×GEO×A-**set**×(T|(T×TI×TI)|(T×T)) → attr_Sum
79 attr_A_i(pta,geo,atrs,t **or** (t,ti,ti′) **or** (t,t′)) **as** summary
79 **pre**: is_within(pta,attr_Area(geo))
type
80 attr_Sum = ...

Area Points and Social-Economic Data

- Similarly as for *Area Points and Geographic Data*.

5.4.1. Etcetera

A first point has been made through the above exemplifying a beginning formal analysis of the "input" arguments for the urban planning functions and behaviours.

MORE TO COME

[23] We shall omit a proper treatment of a topology of areas, borders and points

5.5. Urban Planning Project Management

In this research note we have focused on the urban planning project behaviours, their interactions, and their information "passing". Usually publications about urban planning: research papers, technical papers, survey papers, etcetera, focus on specific "functions". In this research note we do not. Such "functions" are, in this note, embodied in

- b_tri_fit (formula Item 28 on Page 12),
- base_up_fct (formula Item 39 on Page 13),
- base_satisfactory (formula Item 44 on Page 13),
- der_satisfactory$_i$ (formula Item 44 on Page 13) and
- der_up_fct$_i$ (formula Item 57 on Page 16).

We focus instead on what we can say about the domain of urban planning: the fact, or the possibility, that an initial, a core, here referred to as a base, urban planning effort (i.e., project, hence behaviour) can "spew off", generate, a number of (derived, i.e., in some sense subsidiary), more specialised, urban planning projects.

5.5.1. Urban Planning Projects

We think of a comprehensive urban planning project as carried out by urban planners. As is evident from the model given in Sects. 3–4, the project consists of one base urban planning project and up to n derived urban planning projects. The urban planners involved in these projects are professionals in the areas of planning:

- base urban planning issues:
  - ◇ geodesy,
  - ◇ geotechniques,
  - ◇ meteorology,
- base urban plans:
  - ◇ cartography,
  - ◇ cadestral matters,
  - ◇ zoning;
- derived urban planning issues:
  - ◇ industries,
  - ◇ residential and shopping,
  - ◇ apartment buildings,
  - ◇ villaes,
  - ◇ recreational,
  - ◇ etcetera;
- technological infrastructures:
  - ◇ transport,
  - ◇ electricity,
  - ◇ telecommunications,
  - ◇ gas,
  - ◇ water,
  - ◇ waste,
  - ◇ etcetera;
- societal infrastructures:
  - ◇ health care,
  - ◇ schools,
  - ◇ police,
  - ◇ fire brigades,
  - ◇ etcetera;
- etcetera, etcetera, etcetera !

To anyone with any experience in getting such diverse groups and individuals of highly skilled professionals to work together it is obvious that some form of management is required. The term 'comprehensive' was mentioned above. It is meant to express that the comprehensive urban planning project is the only one "dealing" with a given geographic area, and that no other urban planning projects "infringe" upon it, that is, "deal" with sub-areas of that given geographic area.

5.5.2. Strategic, Tactical and Operational Management

We can distinguish between

- strategic,
- tactical and
- operational

management.

Project Resources But first we need take a look at the **resources** that management is charged with:

- the urban planners, i.e., humans,
- time,
- finances,
- office space,
- support technologies: computing etc.,
- etcetera.

Strategic Management By **strategic management** we shall understand the analysis and decisions of, and concerning, scarce resources: people (skills), time, monies: their deployment and trade-offs.

Tactical Management By **tactical management** we shall understand the analysis and decisions with respect to budget and time plans, and the monitoring and control of serially reusable resources: office space, computing.

Operational Management By **operational management** we shall understand the monitoring and control of the enactment, progress and completion of individual deliverables, i.e., documents, the quality (adherence to "standards", fulfillment of expectations, etc.) of these documents, and the day-to-day human relations.

5.5.3. Urban Planning Management

The above (*strategic, tactical & operational management*) translates, in the context of *urban planning*, into:

MORE TO COME

6. Conclusion

TO BE WRITTEN

7. Bibliography

7.1. Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [10, Part IV] — with [5, 2] being exceptions: [12] suggests a number of domain science and engineering research topics; [22] covers the concept of domain facets; [48] explores compositionality and Galois connections. [13, 47] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [25] takes the triptych software development as a basis for outlining principles for believable software management; [17, 32] presents a model for Stanisław Leśniewski's [53] concept of mereology; [23, 26] present an extensive example and is otherwise a precursor for the present paper; [27] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [29] analyses the `TripTych`, especially its domain engineering approach, with respect to [59, 60, Maslow]'s and [61, Peterson's and Seligman's]'s notions of humanity: how can computing relate to notions of humanity; the first part of [34] is a precursor for [40] with the second part of [34] presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in [40]; and with [35] focus on domain safety criticality. The published paper [40] now constitutes the base introduction to domain science & engineering.

7.2. Domain Modeling Experiments

- Credit Card System[24], [36] 2016. Result of my PhD lectures at Uppsala, May 2016
- Weather Information Systems[25] [42] Result of my PhD lectures at Bergen, November 2016
- Documents[26] [28] 2013.
- Transport Systems[27] [31] 2010.

[24] http://www.imm.dtu.dk/ dibj/2016/uppsala/accs.pdf
[25] http://www.imm.dtu.dk/ dibj/2016/wis/wis-p.pdf
[26] http://www.imm.dtu.dk/~dibj/doc-p.pdf
[27] http://www.imm.dtu.dk/~dibj/comet/comet1.pdf

- The Tokyo Stock Exchange Trading Rules[28] and[29] [44] 2010.
- On Development of Web-based Software[30] 2010.
- What is Logistics ?[31] [18] 2009.
- Pipelines – a Domain Description[32] and[33], [30] 2009.
- Platooning[34],
- A Container Line Industry Domain[35], [11] 2007
- Models of IT Security: Security Rules & Regulations[36] [45] 2006.
- Markets[37] [4]
- **Railway Systems Descriptions: 1996–2003**
  - ⋄ Dines Bjørner: Formal Software Techniques in Railway Systems[38] [3]
  - ⋄ Chris George, Dines Bjørner and Søren Prehn: Scheduling and Rescheduling of Trains[39], [49] 1996
  - ⋄ Dines Bjørner: A Railway Systems Domain[40] An "old" UNU-IIST report, 1997
  - ⋄ Dines Bjørner: Formal Software Techniques in Railway Systems[41], 2002
  - ⋄ Albena Strupchanska, Martin Penicka and Dines Bjørner: Railway Staff Rostering[42], 2003 [64]
  - ⋄ Dines Bjørner: Dynamics of Railway Nets[43], 2003 [6]
  - ⋄ Martin Penicka, Albena Strupchanska and Dines Bjørner: Train Maintenance Routing[44], 2003 [63]
  - ⋄ Panagiotis Karras and Dines Bjørner: Train Composition and Decomposition: Domain and Requirements[45], 2003
  - ⋄ Dines Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering[46] [6] 2003

7.3. References

[1] Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[3] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.

[4] Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version.

[5] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[6] Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki. Final version.

[7] Dines Bjørner. Towards a Formal Model of CyberRail. In *Building the Information Society, IFIP 18th World Computer Congress, Tpical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart*, pages 657–664. Kluwer Academic Publishers, August 2004. Original report also listed some of DB's students as co–authors.

[8] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and*

[28] http://www.imm.dtu.dk/~dibj/todai/tse-1.pdf
[29] http://www.imm.dtu.dk/~dibj/todai/tse-2.pdf
[30] http://www.imm.dtu.dk/~dibj/wfdftp.pdf
[31] http://www.imm.dtu.dk/~dibj/logistics.pdf
[32] http://www.imm.dtu.dk/~dibj/pipeline.pdf
[33] http://www.imm.dtu.dk/~dibj/pipe-p.pdf
[34] http://www.imm.dtu.dk/~dibj/platoon-p.pdf
[35] http://www.imm.dtu.dk/~dibj/container-paper.pdf
[36] http://www.imm.dtu.dk/~dibj/it-security.pdf
[37] http://www2.imm.dtu.dk/~db/themarket.pdf
[38] http://www2.compute.dtu.dk/~dibj/rails.pdf
[39] http://www.imm.dtu.dk/ dibj/amore/docs/scheduling.pdf
[40] http://www.imm.dtu.dk/ dibj/UNU-IIST-railways.pdf
[41] http://www.imm.dtu.dk/ dibj/amore/docs/dines-ifac.pdf
[42] http://www.imm.dtu.dk/ dibj/amore/docs/albena-amore.pdf
[43] http://www.imm.dtu.dk/ dibj/amore/docs/ifac-dynamics.pdf
[44] http://www.imm.dtu.dk/ dibj/amore/docs/martin-amore.pdf
[45] http://www.imm.dtu.dk/ dibj/amore/docs/panos-amore.pdf
[46] http://www2.imm.dtu.dk/~db/ifac-dynamics.pdf

Languages; Vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[9] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.

[10] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[11] Dines Bjørner. A Container Line Industry Domain. Techn. report, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. Extensive Draft.

[12] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[13] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[14] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

[15] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.

[16] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.

[17] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[18] Dines Bjørner. What is Logistics ? A Domain Analysis. Techn. report, Incomplete Draft, Fredsvej 11, DK-2840 Holte, Denmark, June 2009.

[19] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[20] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[21] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[22] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[23] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[24] Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Fredsvej 11, DK-2840 Holte, Denmark, January, February 2010.

[25] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[26] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[27] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[28] Dines Bjørner. Documents – a Domain Description[47]. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[29] Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[30] Dines Bjørner. Pipelines – a Domain Description[48]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[31] Dines Bjørner. Road Transportation – a Domain Description[49]. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[32] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[33] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. (paper[50], slides[51]).

[34] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José

[47] http://www.imm.dtu.dk/~dibj/doc-p.pdf
[48] http://www.imm.dtu.dk/~dibj/pipe-p.pdf
[49] http://www.imm.dtu.dk/~dibj/road-p.pdf
[50] http://www.imm.dtu.dk/~dibj/jaist-da.pdf
[51] http://www.imm.dtu.dk/~dibj/jaist-s.pdf

Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[35] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[36] Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/~dibj/2016/credit/accs.pdf.

[37] Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. *Submitted for consideration to Formal Aspects of Computing*, 2016. http://www.imm.dtu.dk/~dibj/2016/process/process-p.pdf.

[38] Dines Bjørner. Domain Facets: Analysis & Description. *Submitted for consideration to Formal Aspects of Computing*, 2016. http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf.

[39] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, 2016. http://www.imm.dtu.dk/~dibj/2016/demos/-faoc-demo.pdf.

[40] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, ...(...):1–51, 2016. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/10.1007/s00165-016-0385-z.

[41] Dines Bjørner. To Every Manifest Domain a `CSP` Expression — A Rôle for Mereology in Computer Science. Submitted for consideration to Journal of Logical and Algebraic Methods in Programming, Fredsvej 11, DK–2840 Holte, Denmark, December 2016. http://www.imm.dtu.dk/~dibj/2016/mereo/mereo.pdf.

[42] Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/~dibj/2016/wis/wis-p.pdf.

[43] Dines Bjørner. What are Documents ? Research Note, July 2017. http://www.imm.dtu.dk/~dibj/2017/docs/-docs.pdf.

[44] Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Fredsvej 11, DK-2840 Holte, Denmark, January and February, 2010. Version 1, 78 pages: many auxiliary appendices, Version 2, 23 pages: omits many appendices and corrects some errors..

[45] Dines Bjørner. *[46] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[46] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.

[47] Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[48] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[49] Dines Bjørner, Chris W. George, and Søren Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice*, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.

[50] Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science. Extended version.

[51] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[52] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[53] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[54] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[55] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[56] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[57] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[58] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[59] Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. http://psych-classics.yorku.ca/Maslow/motivation.htm.

[60] Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.

[61] Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.

[62] Martin Pěnička and Dines Bjørner. From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In *Building the Information Society, IFIP 18th World Computer Congress,*

Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart, pages 629–636. Kluwer Academic Publishers, August 2004.

[63] Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version.

[64] Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version.

[65] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

A. A Document System

A.1. Introduction

We analyse a notion of documents. Documents such as they occur in daily life. What can we say about documents – regardless of whether we can actually provide compelling evidence for what we say! That is: we model documents, not as electronic entities — which they are becoming, more-and-more, but as if they were manifest entities. When we, for example, say that *"this document was recently edited by such-and-such and the changes of that editing with respect to the text before is such-and-such"*, then we can, of course, always claim so, even if it may be difficult or even impossible to verify the claim. It is a fact, although maybe not demonstrably so, that there was a version of any document before an edit of that document. It is a fact that some handler did the editing. It is a fact that the editing took place at (or in) exactly such-and-such a time (interval), etc. We model such facts.

A.2. A Document Systems Description

This research note unravels its analysis &[52] description in stages.

A.3. A System for Managing, Archiving and Handling Documents

The title of this section: *A System for Managing, Archiving and Handling Documents* immediately reveals the major concepts: That we are dealing with a *system* that **manages**, **archives** and **handles documents.** So what do we mean by **managing, archiving** and **handling** documents, and by **documents** ? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are! **Management** decides[53] to direct **handler**s to work on **document**s. **Management** first directs the document archive to **create document**s. The document **archive creates document**s, as requested by **management**, and informs management of the **unique document identifiers** (by means of which handlers can handle these documents). **Management** then **grant**s its designated **handler**(s) **access rights** to **document**s, these access rights enable handlers to **edit, read** and **copy** documents. The **handler**s' **edit**ing and **read**ing of **document**s is accomplished by the **handler**s "working directly" with the **document**s (i.e., synchronising and communicating with **document behaviour**s). The **handler**s' **copy**ing of **document**s is accomplished by the **handler**s requesting **management**, in collaboration with the **archive** behaviour, to do so.

A.4. Principal Endurants

By an *endurant* we shall understand *"an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time."* Were we to "freeze" time we would still be able to observe the entire endurant. This characterisation of what we mean by an 'endurant' is from [40, Manifest Domains: Analysis & Description]. We begin by identifying the principal endurants.

81 From document handling systems one can observe aggregates of handlers and documents.
   We shall refer to 'aggregates of handlers' by M, for management, and to 'aggregates of documents' by A, for archive.

82 From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).

83 From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

type
81 S, M, A
value
81 obs_M: S → M

[52] We use the logogram & between two terms, A & B, when we mean to express one meaning.
[53] How these decisions come about is not shown in this research note – as it has nothing to do with the essence of document handling, but, perhaps, with 'management'.

81 obs_A: S → A
type
82 H, Hs = H-**set**
83 D, Ds = D-**set**
value
82 obs_Hs: M → Hs
83 obs_Ds: A → Ds

A.5. Unique Identifiers

The notion of unique identifiers is treated, at length, in [40, Manifest Domains: Analysis & Description].

84 We associate unique identifiers with aggregate, handler and document endurants.
85 These can be observed from respective parts[54].

type
84 MI[55], AI[56], HI, DI
value
85 uid_MI[57]: M → MI
85 uid_AI[58]: A → AI
85 uid_HI: H → HI
85 uid_DI: D → DI

As reasoned in [40, Manifest Domains: Analysis & Description], the unique identifiers of endurant parts are indeed unique: No two parts, whether composite, as are the aggregates, or atomic, as are handlers and documents, can have the same unique identifiers.

A.6. Documents: A First View

A document is a written, drawn, presented, or memorialized representation of thought. The word originates from the Latin documentum, which denotes a "teaching" or "lesson".[59] We shall, for this research note, take a document in its written and/or drawn form. In this section we shall survey the concept a documents.

A.6.1. Document Identifiers

Documents have *unique identifiers.* If two or more documents have the same document identifier then they are the same, one (and not two or more) document(s).

A.6.2. Document Descriptors

With documents we associate *document descriptors*. We do not here stipulate what document descriptors are other than saying that when a document is **create**d it is provided with a descriptor and this descriptor "remains" with the document and never changes value. In other words, it is a static attribute.[60] We do, however, include, in document descriptors, that the document they describe was initially based on a set of zero, one or more documents – identified by their unique identifiers.

[54] [40, Manifest Domains: Analysis & Description] explains how 'parts' are the discrete endurants with which we associate the full complement of properties: unique identifiers, mereology and attributes.
[55] We shall not, in this research note, make use of the (one and only) management identifier.
[56] We shall not, in this research note, make use of the (one and only) archive identifier.
[57] Cf. Footnote 55: hence we shall not be using the uid_MI observer.
[58] Cf. Footnote 56: hence we shall not be using the uid_AI observer.
[59] From: https://en.wikipedia.org/wiki/Document
[60] You may think of a document descriptor as giving the document a title; perhaps one or more authors; perhaps a physical address (of, for example, these authors); an initial date; as expressing whether the document is a research, or a technical report, or other; who is issuing the document (a public institution, a private firm, an individual citizen, or other); etc.

A.6.3. Document Annotations

With documents we also associate *document annotations*. By a document annotation we mean a programmable attribute, that is, an attribute which can be 'augmented' by document handlers. We think of document annotations as "incremental", that is, as "adding" notes "on top of" previous notes. Thus we shall model document annotations as a repository: notes are added, i.e., annotations are augmented, previous notes are not edited, and no notes are deleted. We suggest that notes be time-stamped. The notes (of annotations) may be such which record handlers work on documents. Examples could be: *"10 September 2017: 05:36 am: This is version V."*, *"This document was released on 10 September 2017: 05:36 am."*, *"10 September 2017: 05:36 am: Section X.Y.Z of version III was deleted."*, *"10 September 2017: 05:36 am: References to documents* doc_i *and* doc_j *are inserted on Pages* p *and* q*, respectively."* and *"10 September 2017: 05:36 am: Final release."*

A.6.4. Document Contents: Text/Graphics

The main idea of a document, to us, is the *written* (i.e., text) and/or *drawn* (i.e., graphics) *contents*. We do not characterise any format for this *contents*. We may wish to insert, in the *contents*, references to locations in the *contents* of other documents. But, for now, we shall not go into such details. The main operations on documents, to us, are concerned with: their **creation, editing, reading, copying** and **shredding**. The **editing** and **reading** operations are mainly concerned with document *annotations* and *text/graphics*.

A.6.5. Document Histories

So documents are **create**d, **edit**ed, **read**, **copied** and **shred**ed. These operations are initiated by the management (**create**), by the archive (**create**), and by handlers (**edit, read, copy**), and at specific times.

A.6.6. A Summary of Document Attributes

86 As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.
87 Document annotations are lists of document notes.
88 Document histories are lists of time-stamped document operation designators.
89 A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.
90 A create designator identifies
   a a handler and a time (at which the create request first arose), and presents
   b elements for constructing a document descriptor, one which
      i besides some further undefined information
      ii refers to a set of documents (i.e., embeds reference to their unique identifiers),
   c a (first) document note, and
   d an empty document contents.
91 An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.
92 A read designator identifies a handler.
93 A copy designator identifies a handler, a time, the document to be copied (by its unique identifier, and a document note to be inserted in both the master and the copy document.
94 A shred designator identifies a handler.
95 An edit function takes a triple of a document annotation, a document note and document contents and yields a pair of a document annotation and a document contents.
96 An undo function takes a pair of a document note and document contents and yields a triple of a document annotation, a document note and a document contents.
97 Proper pairs of (edit,undo) functions satisfy some inverse relation.

There is, of course, no need, in any document history, to identify the identifier of that document.

```
type
86  DD, DA, DC, DH
value
86  attr_DD: D → DD
86  attr_DA: D → DA
86  attr_DC: D → DC
86  attr_DH: D → DH
type
87  DA = DN*
88  DH = (TIME × DO)*
89  DO == Crea | Edit | Read | Copy | Shre
90  Crea :: (HI × TIME) × (DI-set × Info) × DN × {|"empty_DC"|}
90bi  Info = ...
value
90bii  embed_DIs_in_DD: DI-set × Info → DD
axiom
90d  "empty_DC" ∈ DC
type
91  Edit :: (HI × TIME) × (EDIT × UNDO)
92  Read :: (HI × TIME) × DI
93  Copy :: (HI × TIME) × DI × DN
94  Shre :: (HI × TIME) × DI
95  EDIT = (DA × DN × DC) → (DA × DC)
96  UNDO = (DA × DC) → (DA × DN × DC)
axiom
97  ∀ mkEdit(_,(e,u)):Edit •
97    ∀ (da,dn,dc):(DA×DN×DC) •
97      u(e(da,dn,dc))=(da,dn,dc)
```

A.7. Behaviours: An Informal, First View

In [40, Manifest Domains: Analysis & Description] we show that we can associate behaviours with parts, where parts are such discrete endurants for which we choose to model all its observable properties: unique identifiers, mereology and attributes, and where behaviours are sequences of actions, events and behaviours.

- The overall document handler system behaviour can be expressed in terms of the parallel composition of the behaviours

  98 of the system core behaviour,
  99 of the handler aggregate (the management) behaviour
  100 and the document aggregate (the archive) behaviour,

  with the (distributed) parallel composition of

  101 all the behaviours of handlers and,

  the (distributed) parallel composition of

  102 at any one time, zero, one or more behaviours of documents.

- To express the latter

  103 we need introduce two "global" values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

```
value
103  his:HI-set, dis:DI-set
```

98 sys(...)
99 ‖ mgtm(...)
100 ‖ arch(...)
101 ‖ ‖{hdlr$_i$(...)|i:HI•i∈his}
102 ‖ ‖{docu$_i$(dd)(da,dc,dh)|i:DI•i∈dis}

For now we leave undefined the arguments, (...) etc., of these behaviours. The arguments of the document behaviour, (dd)(da,dc,dh), are the static, respectively the three programmable (i.e., dynamic) attributes: *document descriptor, document annotation, document contents* and *document history*. The above expressions, Items 99–102, do not define anything, they can be said to be "snapshots" of a "behaviour state". Initially there are no document behaviours, docu$_i$(dd)(da,dc,dh), Item 102. Document behaviours are "started" by the archive behaviour (on behalf of the management and the handler behaviours). Other than mentioning the system (core) behaviour we shall not model that behaviour further.

A.8. Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).

104 The management behaviour, mgtm, need to (synchronise and) communicate with the archive behaviour, arch, in order, for the management behaviour, to request the archive behaviour
- to **create** (ab initio or due to **copy**ing)
- or **shred** document behaviours, docu$_j$,

and for the archive behaviour
- to inform the management behaviour of the identity of the document(behaviour)s that it has created.

channel
104 mgtm_arch_ch:MA

105 The management behaviour, mgtm, need to (synchronise and) communicate with all handler behaviours, hdlr$_i$ and they, in turn, to (synchronised) communicate with the handler management behaviour, mgtm. The management behaviour need to do so in order
- to inform a handler behaviour that it is granted access rights to a specific document, subsequently these access rights may be modified, including revoked.

channel
105 {mgtm_hdlr_ch[i]:MH|i:HI•i ∈ his}

106 The document archive behaviour, arch, need (synchronise and) communicate with all document behaviours, docu$_j$ and they, in turn, to (synchronise and) communicate with the archive behaviour, arch.

channel
106 {arch_docu_ch[j]:AD|h:DI•j ∈ dis}

107 Handler behaviours, hdlr$_i$, need (synchronise and) communicate with all the document behaviours, docu$_j$, with which it has operational allowance to so do so[61], and document behaviours, docu$_j$, need (synchronise and) communicate with potentially all handler behaviours, hdlr$_i$, namely those handler behaviours, hdlr$_i$ with which they have ("earlier" synchronised and) communicated.

channel
107 {hdlr_docu_ch[i,j]:HD|i:HI,j:DI•i ∈ his∧j ∈ dis}

[61] The notion of operational allowance will be explained below.

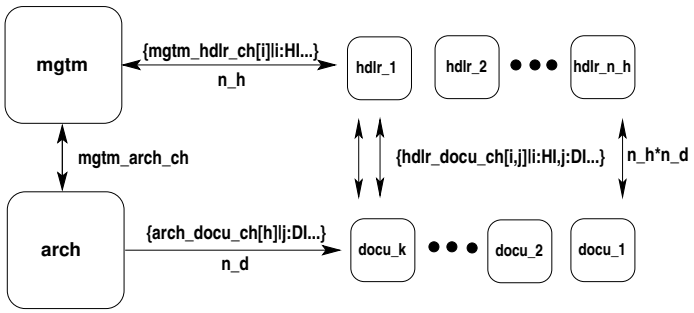

Fig. 3. An Informal Snapshot of System Behaviours

108 At present we leave undefined the type of messages that are communicated.

type
108 MA, MH, AD, HD

A.9. An Informal Graphical System Rendition

Figure 3 is an informal rendition of the "state" of a number of behaviours: a single management behaviour, a single archive behaviour, a fixed number, n_h, of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being n_d. The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; n_h channels (n_h is, again, informally indicated) between the management behaviour and the n_h handler behaviours; n_d channels (n_d is, again, informally indicated) between the archive behaviour and the n_d document behaviours; and $n_h \times n_d$ channels ($n_d \times n_d$ is, again, informally indicated) between the n_h handler behaviours and the n_d document behaviours

A.10. Behaviour Signatures

109 The mgtm behaviour (synchronises and) communicates with the archive behaviour and with all of the handler behaviours, hdlr_i.

110 The archive behaviour (synchronises and) communicates with the mgtm behaviour and with all of the document behaviours, docu_j.

111 The signature of the generic handler behaviours, hdlr_i expresses that they [occasionally] receive "orders" from management, and otherwise [regularly] interacts with document behaviours.

112 The signature of the generic document behaviours, docu_j expresses that they [occasionally] receive "orders" from the archive behaviour and that they [regularly] interacts with handler behaviours.

value
109 mgtm: ... → **in,out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
110 arch: ... → **in,out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
111 hdlr_i: ... → **in** mgtm_hdlr_ch[i], **in,out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
112 docu_j: ... → **in** mgtm_arch_ch, **in,out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **Unit**

A.11. Time

A.11.1. Time and Time Intervals: Types and Functions

113 We postulate a notion of time, one that covers both a calendar date (from before Christ up till now and beyond). But we do not specify any concrete type (i.e., format such as: YY:MM:DD, HH:MM:SS).
114 And we postulate a notion of (signed) time interval — between two times (say: ±YY:MM:DD:HH:MM:SS).
115 Then we postulate some operations on time: Adding a time interval to a time obtaining a time; subtracting one time from another time obtaining a time interval, multiplying a time interval with a natural number; etc.
116 And we postulate some relations between times and between time intervals.

```
type
113 TIME
114 TIME_INTERVAL
value
115 add: TIME_INTERVAL × TIME → TIME
115 sub: TIME × TIME → TIME_INTERVAL
115 mpy: TIM_INTERVALE × Nat → TIME_INTERVAL
116 <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → Bool
```

A.11.2. A Time Behaviour and a Time Channel

117 We postulate a[n "ongoing"] time behaviour: it either keeps being a time behaviour with unchanged time, t, or – internally non-deterministically – chooses being a time behaviour with a time interval incremented time, t+ti, or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, time_ch, then resumes being a time behaviour with unchanged time., t
118 The time interval increment, ti, is likewise internally non-deterministically chosen. We would assume that the increment is "infinitesimally small", but there is no need to specify so.
119 We also postulate a channel, time_ch, on which the time behaviour offers time values to whoever so requests.

```
value
117  time: TIME → time_ch  TIME  Unit
117  time(t) ≡ (time(t) ⌈⌈ time(t+ti) ⌈⌈ time_ch!t ; time(t))
118  ti:TIME_INTERVAL ...
channel
119  time_ch:TIME
```

A.11.3. An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [56] Specification Language, RSL [55]. We shall be making use of an informal language construct:

- **wait** ti.

wait is a keyword; ti designates a time interval. A typical use of the wait construct is:

- ... *ptA* ; **wait** ti; *ptB* ; ...

If at specification text point *ptA* we may assert that time is t, then at specification text point *ptB* we can assert that time is t+ti.

A.12. Behaviour "States"

We recall that the endurant parts, Management, Archive, Handlers, and Documents, have properties in the form of *unique identifiers*, *mereologies* and *attributes*. We shall not, in this research note, deal with

possible mereologies of these endurants. In this section we shall discuss the endurant attributes of mgtm (management), arch (archive), hdlrs (handlers), and docus (documents). Together the values of these properties, notably the attributes, constitute states – and, since we associate behaviours with these endurants, we can refer to these states also a behaviour states. Some attributes are static, i.e., their value never changes. Other attributes are dynamic.[62] Document handling systems are rather conceptual, i.e., abstract in nature. The dynamic attributes, therefore, in this modeling "exercise", are constrained to just the *programmable* attributes. Programmable attributes are those whose value is set by "their" behaviour. For a behaviour β we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

value β: Static → Program → ... **Unit**

120 For the management endurant/behaviour we focus on one programmable attribute. The management behaviour needs keep track of all the handlers it is charged with, and for each of these which zero, one or more documents they have been granted access to (cf. Sect. A.13.3 on the facing page). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.

121 For the archive behaviour we similarly focus on one programmable attribute. The archive behaviour needs keep track of all the documents it has used (i.e., created), those that are avaliable (and not yet used), and of those it has shredded. Initially all these three archive directory sets are empty.

122 For the handler behaviour we similarly focus on one programmable attribute. The handler behaviour needs keep track of all the documents it has been charged with and its access rights to these.

123 Document attributes we mentioned above, cf. Items 86–89.

type
120 MDIR = HI $\overrightarrow{m}$ (DI $\overrightarrow{m}$ ANm-**set**)
121 ADIR = avail:DI-**set** × used:DI-**set** × gone:DI-**set**
122 HDIR = DI $\overrightarrow{m}$ ANm-**set**
123 SDATR = DD, PDATR = DA × DC × DH
axiom
121 ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used

We can now "complete" the behaviour signatures. We omit, for now, static attributes.

value
109 mgtm: MDIR → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
110 arch: ADIR → **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
111 hdlr$_i$: HDIR → **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
112 docu$_j$: SDATR → PDATR → **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **Unit**

A.13. Inter-Behaviour Messages

Documents are not "fixed, innate" entities. They embody a "history", they have a "past". Somehow or other they "carry a trace of all the "things" that have happened/occurred to them. And, to us, these things are the manipulations that management, via the archive and handlers perform on documents.

A.13.1. Management Messages with Respect to the Archive

124 Management **create** documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document "state" and start a document behaviour, with initial information, cf. Item 90 on Page 31:

a the identity of the initial handler of the document to be created,
b the time at wich the request is being made,

[62] We refer to Sect. 3.4 of [40], and in particular its subsection 3.4.4.

c a document descriptor which embodies a (finite) set of zero or more (used) document identifiers (dis),
d a document annotation note dn, and
e an initial, i.e., "empty" contents, "empty_DC".

type
90. Crea :: (HI × TIME) × (DI-**set** × Info) × DN × {|"empty_DC"|} [cf. formula Item 90, Page 32]

125 The management behaviour passes on to the archive behaviour, requests that it accepts from handlers behaviours, for the copying of document:

125 Copy :: DI × HI × TIME × DN [cf. Item 135 on the following page]

126 Management **schred**s documents by informing the archive behaviour to do so.

type
126 Shred :: TIME × DI

A.13.2. Management Messages with Respect to Handlers

127 Upon receiving, from the archive behaviour, the "feedback" the identifier of the created document (behaviour):

type
127. Create_Reply :: NewDocID(di:DI)

128 the management behaviour decides to **grant** access rights, acrs:ACRS[63], to a document handler, hi:HI.

type
128 Gran :: HI × TIME × DI × ACRS

A.13.3. Document Access Rights

Implicit in the above is a notion of document access rights.

129 By document access rights we mean a set of action names.
130 By an action name we mean such tokens that indicate either of the document handler operations indicate above.

type
129 ACRS = ANm-**set**
130 ANm = {|"edit","read","copy"|}

A.13.4. Archive Messages with Respect to Management

To create a document management provides the archive with some initial information. The archive behaviour selects a document identifier that has not been used before.

131 The archive behaviour informs the management behaviour of the identifier of the created document.

type
131 NewDocID :: DI

[63] For the concept of access rights see Sect. A.13.3.

A.13.5. Archive Message with Respect to Documents

132 To shred a document the archive behaviour must access the designated document in order to **stop** it. No "message", other than a symbolic `"stop"`, need be communicated to the document behaviour.

type
132 Shred :: {|"stop"|}

A.13.6. Handler Messages with Respect to Documents

Handlers, generically referred to by hdlr_i, may perform the following operations on documents: **edit, read** and **copy**. (Management, via the archive behaviour, **create**s and **shred**s documents.)

133 To perform an **edit** action handler hdlr_i must provide the following:

- the document identity – in the form of a (i:HI,j:DI) channel hdlr_docu_ch index value,
- the handler identity, i,
- the time of the edit request,
- and a pair of functions: one which performs the editing and one which un-does it!

type
133 Edit :: DI × HI × TIME × (EDIT × UNDO)

134 To perform a **read** action handler hdlr_i must provide the following information:

- the document identity – in the form of a di:DI channel hdlr_docu_ch index value,
- the handler identity and
- the time of the read request.

type
134 Read :: DI × HI × TIME

A.13.7. Handler Messages with Respect to Management

135 To perform a **copy** action, a handler, hdlr_i, must provide the following information to the management behaviour, mgtm:

- the document identity,
- the handler identity – in the form of an hi:HI channel mgtm_hdlr_ch index value,
- the time of the copy request, and
- a document note (to be affixed both the master and the copy documents).

135 Copy :: DI × HI × TIME × DN [cf. Item 125 on the preceding page]

How the handler, the management, the archive and the "named other" handlers then enact the copying, etc., will be outlined later.

A.13.8. A Summary of Behaviour Interactions

Figure 4 on the next page summarises the sources, **out**, resp. !, and the targets, **in**, resp. ?, of the messages covered in the previous sections.

A.14. A General Discussion of Handler and Document Interactions

We think of documents being manifest. Either a document is in paper form, or it is in electronic form. In paper form we think of a document as being in only one – and exactly one – physical location. In electronic

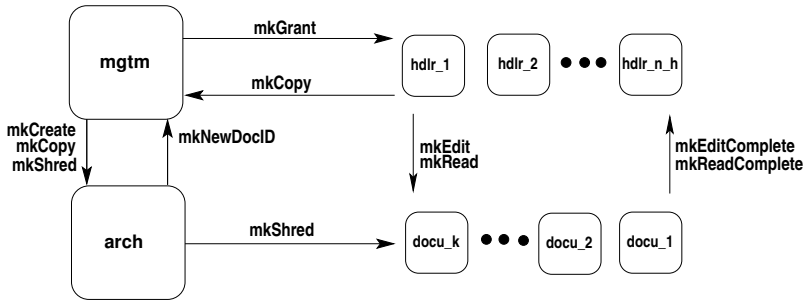

Fig. 4. A Summary of Behaviour Interactions

form a document is also in only one – and exactly one – physical location. No two handlers can access the same document at the same time or in overlapping time intervals. If your conventional thinking makes you think that two or more handlers can, for example, read the same document "at the same time", then, in fact, they are reading either a master and a copy of that master, or they are reading two copies of a common master.

A.15. Channels: A Final View

We can now summarize the types of the various channel messages first referred to in Items 104, 105, 106 and 107.

type
104 MA = Create (Item 124 on Page 36) | Shred (Item 124d on Page 37) | NewDocID (Item 131 on Page 37)
105 MH = Grant (Item 124c on Page 37) | Copy (Item 135 on the facing page) |
106 AD = Shred (Item 132 on the preceding page)
107 HD = Edit (Item 133 on the facing page) | Read (Item 134 on the preceding page) | Copy (Item 135 on the facing page)

A.16. An Informal Summary of Behaviours

A.16.1. The Create Behaviour: Left Fig. 5 on the next page

136 [1] The management behaviour, at its own volition, initiates a create document behaviour. It does so by offering a create document message to the archive behaviour.

- a [1.1] That message contains a meaningful document descriptor,
- b [1.2] an initial document annotation,
- c [1.3] an "empty" document contents and
- d [1.4] a single element document history.

(We refer to Sect. A.13.1 on Page 36, Items 124–124e.)

137 [2] The archive behaviour offers to accept that management message. It then selects an available document identifier (here shown as k), henceforth marking k as used.

138 [3] The archive behaviour then "spawns off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.

139 [4] The archive behaviour then offers the document identifier k message to the management behaviour. (We refer to Sect. A.13.4 on Page 37, Item 131.)

140 [5] The management behaviour then

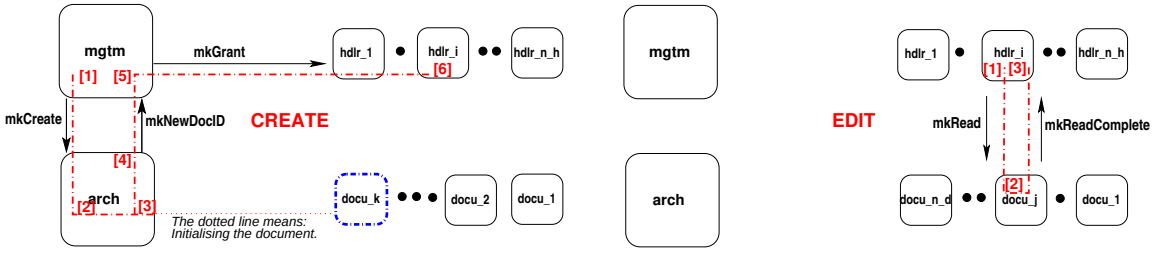

Fig. 5. Informal Snapshots of Create and Edit Document Behaviours

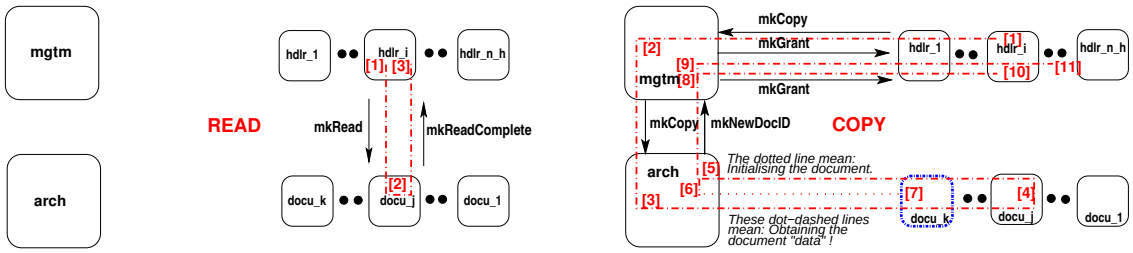

Fig. 6. Informal Snapshots of Read and Copy Document Behaviours

a [5.1] selects a handler, here shown as i, i.e., hdlr_i,
b [5.2] records that that handler is granted certain access rights to document k,
c [5.3] and offers that granting to handler behaviour i.

(We refer to Sect. A.13.2 on Page 37, Item 128 on Page 37.)

141 [6] Handler behaviour i records that it now has certain access rights to doccument i.

A.16.2. The Edit Behaviour: Right Fig. 5

1 Handler behaviour i, at its own volition, initiates an edit action on document j (where i has editing rights for document j). Handler i, optionally, provides document j with a(annotation) note. While editing document j handler i also "selects" an appropriate pair of *edit/undo* functions for document j.

2 Document behaviour j accepts the editing request, enacts the editing, optionally appends the (annotation) note, and, with handler i, completes the editing, after some time interval ti.

3 Handler behaviour i completes its edit action.

A.16.3. The Read Behaviour: Left Fig. 6

1 Handler behaviour i, at its own volition, initiates a read action on document j (where i has reading rights for document j). Handler i, optionally, provides document j with a(annotation) note.

2 Document behaviour j accepts the reading request, enacts the reading by providing the handler, i, with the document contents, and optionally appends the (annotation) note, and, with handler i, completes the reading, after some time interval ti.

3 Handler behaviour i completes its read action.

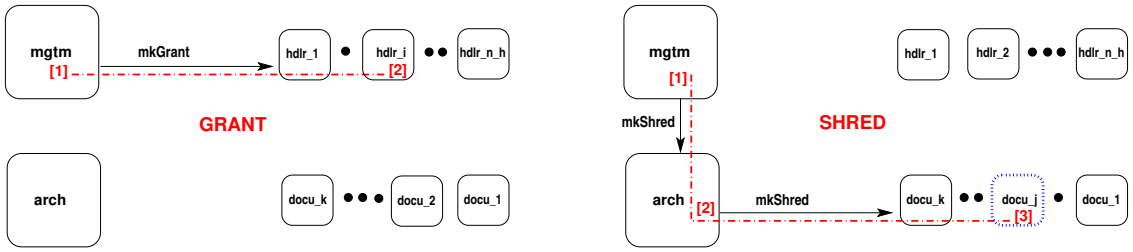

Fig. 7. Informal Snapshots of Grant and Shred Document Behaviours

A.16.4. The Copy Behaviour: Right Fig. 6 on the facing page

1 Handler behaviour i, at its own volition, initiates a copy action on document j (where i has copying rights for document j). Handler i, optionally, provides master document j as well as the copied document (yet to be identified) with respective (annotation) notes.
2 The management behaviour offers to accept the handler message. As for the create action, the management behaviour offers a combined *copy and create* document message to the archive behaviour.
3 The archive behaviour selects an available document identifier (here shown as k), henceforth marking k as used.
4 The archive behaviour then obtains, from the master document j its *d*ocument *d*escriptor, dd_j, its *d*ocument *a*nnotations, da_j, its *d*ocument *c*ontents, dc_j, and its document history, dh_j.
5 The archice behaviour informs the management behaviour of the identifier, k, of the (new) document copy,
6 while assembling the attributes for that (new) document copy: its *d*ocument *d*escriptor, dd_k, its *d*ocument *a*nnotations, da_k, its *d*ocument *c*ontents, dc_k, and its document history, dh_k, from these "similar" attributes of the master document j,
7 while then "spawning off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.
8 The management behaviour accepts the identifier, k, of the (new) document copy, recording the identities of the handlers and their access rights to k,
9 while informing these handlers (informally indicated by a "dangling" dash-dotted line) of their grants,
10 while also informing the master copy of the copy identity (etcetera).
11 The handlers granted access to the copy record this fact.

A.16.5. The Grant Behaviour: Left Fig. 7

This behaviour has its

1 Item [1] correspond, in essence, to Item [9] of the copy behaviour – see just above – and
2 Item [2] correspond, in essence, to Item [11] of the copy behaviour.

A.16.6. The Shred Behaviour: Right Fig. 7

1 The management, at its own volition, selects a document, j, to be shredded. It so informs the archive behaviour.
2 The archive behaviour records that document j is to be no longer in use, but shredded, and informs document j's behaviour.
3 The document j behaviour accepts the shred message and **stop**s (indicated by the dotted rounded edge box).

A.17. The Behaviour Actions

To properly structure the definitions of the four kinds of (management, archive, handler and document) behaviours we single each of these out "across" the six behaviour traces informally described in Sects. A.16.1–A.16.6. The idea is that if behaviour β is involved in τ traces, $\tau_1, \tau_2, ..., \tau_\tau$, then behaviour β shall be defined in terms of τ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, ..., \beta_{\tau_\tau}$.

A.17.1. Management Behaviour

142 The management behaviour is involved in the following action traces:

a **create** Fig. 5 on Page 40 Left
b **copy** Fig. 6 on Page 40 Right
c **grant** Fig. 7 on the previous page Left
d **shred** Fig. 7 on the preceding page Right

```
value
142  mgtm: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[ hi ]|hi:HI•hi ∈ his} Unit
142  mgtm(mdir) ≡
142a       mgtm_create(mdir)
142b     ⌈⌉ mgtm_copy(mdir)
142c     ⌈⌉ mgtm_grant(mdir)
142d     ⌈⌉ mgtm_shred(mdir)
```

Management Create Behaviour: Left Fig. 5 on Page 40

143 The **management create** behaviour
144 initiates a create document behaviour (i.e., a request to the archive behaviour),
145 and then awaits its response.

```
value
143  mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[ hi ]|hi:HI•hi ∈ his} Unit
143  mgtm_create(mdir) ≡
144  [1]   let hi = mgtm_create_initiation(mdir) ;      [Left Fig. 5 on Page 40]
145  [5]   mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. 5 on Page 40]
```

The **management create initiation** behaviour

146 selects a handler on behalf of which it requests the document creation,
147 assembles the elements of the create message:

- by embedding a set of zero or more document references, dis, with some information, info, into a document descriptor, adding
- a document note, dn, and
- and initial, that is, empty document contents, `"empty_DC"`,

148 offers such a create document message to the archive behaviour, and
149 yields the identifier of the chosen handler.

```
value
144  mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[ hi ]|hi:HI•hi ∈ his} HI
144  mgtm_create_initiation(mdir) ≡
146            let hi:HI • hi ∈ dom mdir,
147  [1.2–.4]      (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
148  [1.1]      mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,"empty_DC")
149            hi end

147  is_meaningful: DD → MDIR → Bool [left further undefined]
```

The **management create awaits response** behaviour

150 starts by awaiting a reply from the archive behaviour with the identity, *di*, of the document (that that behaviour has created).
151 It then selects suitable access rights,
152 with which it updates its handler/document directory
153 and offers to the chosen handler
154 whereupon it resumes, with the updated management directory, being the management behaviour.

```
value
145  mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
145  mgtm_create_awaits_response(mdir) ≡
150  [5]      let mkNewDocID(di) = mgtm_arch_ch ? in
151  [5.1]    let acrs:ANm-set in
152  [5.2]    let mdir′ = mdir † [hi ↦ [di ↦ acrs]] in
153  [5.3]    mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
154           mgtm(mdir′) end end end
```

Management Copy Behaviour: Right Fig. 6 on Page 40

155 The **management copy** behaviour
156 accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),
157 and then awaits a response from the archive behaviour;
158 after which it grants access rights to handlers to the document copy.

```
value
155  mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
155  mgtm_copy(mdir) ≡
156  [2]  let hi = mgtm_accept_copy_request(mdir) in
157  [8]  let di = mgtm_awaits_copy_response(mdir)(hi) in
158  [9]  mgtm_grant_access_rights(mdir)(di) end end
```

159 The **management accept copy** behaviour non-deterministically externally ([]) awaits a copy request from a[ny] handler (*i*) behaviour –
160 with the request identifying the master document, *j*, to be copied.
161 The management accept copy behaviour forwards (!) this request to the archive behaviour –
162 while yielding the identity of the requesting handler.

```
159.  mgtm_accept_copy_request: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
159.  mgtm_accept_copy_request(mdir) ≡
160.      let mkCopy(di,hi,t,dn) = []{mgtm_hdlr_ch[i]?|i:HI•i ∈ his} in
161.      mgtm_arch_ch ! mkCopy(di,hi,t,dn) ;
161.      hi end
```

The **management awaits copy response** behaviour

163 awaits a reply from the archive behaviour as to the identity of the newly created copy (*di*) of master document *j*.
164 The management awaits copy response behaviour then informs the 'copying-requesting' handler, *hi*, that the copying has been completed and the identity of the copy (*di*) –
165 while yielding the identity, *di*, of the newly created copy.

```
142b.  mgtm_awaits_copy_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
142b.  mgtm_awaits_copy_response(mdir)(hi) ≡
163.   [8]  let mkNewDocID(di) = mgtm_arch_ch ? in
164.        mgtm_hdlr_ch[hi] ! mkCopy(di) ;
165.        di end
```

The **management grants access rights** behaviour

166 selects suitable access rights for a suitable number of selected handlers.
167 It then offers these to the selected handlers.

```
158. mgtm_grant_access_rights: MDIR → DI → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
158. mgtm_grant_access_rights(mdir)(di) ≡
166.     let diarm = [hi↦acrs|hi:HI,arcs:ANm-set• hi ∈ dom mdir∧arcs⊆(diarm(hi))(di)] in
167.     ‖ {mgtm_hdlr_ch[hi]!mkGrant(hi,time_ch?,di,acrs) |
167.         hi:HI,acrs:ANm-set•hi ∈ dom diarm∧acrs⊆(diarm(hi))(di)} end
```

Management Grant Behaviour: Left Fig. 7 on Page 41 The **management grant** behaviour

168 is a variant of the mgtm_grant_access_rights function, Items 166–167.
169 The management behaviour selects a suitable subset of known handler identifiers, and
170 for these a suitable subset of document identifiers from which
171 it then constructs a map from handler identifiers to subsets of access rights.
172 With this the management behaviour then issues appropriate grants to the chosen handlers.

```
type
    MDIR = HI m→ (DI m→ ANm-set)
value
168 mgtm_grant: MDIR → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
168 mgtm_grant(mdir) ≡
169     let his ⊆ dom dir in
170     let dis ⊆ ∪{dom mdir(hi)|hi:HI•hi ∈ his} in
171     let diarm = [hi↦acrs|hi:HI,di:DI,arcs:ANm-set• hi ∈ his∧di ∈ dis∧acrs⊆(diarm(hi))(di)] in
172     ‖{mgtm_hdlr_ch[hi]!mkGrant(di,acrs) |
172       hi:HI,di:DI,acrs:ANm-set•hi ∈ dom diarm∧di ∈ dis∧acrs⊆(diarm(hi))(di)}
168     end end end
```

Management Shred Behaviour: Right Fig. 7 on Page 41 The **management shred** behaviour

173 initiates a request to the archive behaviour.
174 First the management shred behaviour selects a document identifier (from its directory).
175 Then it communicates a shred document message to the archive behaviour;
176 then it notes the (to be shredded) document in its directory
177 whereupon the management shred behaviour resumes being the management behaviour.

```
value
173  mgtm_shred: MDIR → out mgtm_arch_ch  Unit
173  mgtm_shred(mdir) ≡
174      let di:DI • is_suitable(di)(mdir) in
175  [1] mgtm_arch_ch ! mkShred(time_ch?,di) ;
176      let mdir′ = [hi↦mdir(hi)\{di}|hi:HI•hi ∈ dom mdir] in
177      mgtm(mdir′) end end
```

A.17.2. Archive Behaviour

178 The archive behaviour is involved in the following action traces:

a **create** Fig. 5 on Page 40 Left
b **copy** Fig. 6 on Page 40 Right
c **shred** Fig. 7 on Page 41 Right

```
type
121  ADIR = avail:DI-set × used:DI-set × gone:DI-set
axiom
121  ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
value
178  arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[ di ]|di:DI•di ∈ dis} Unit
178a  arch(adir) ≡
178a        arch_create(adir)
178b      ⌈⌉ arch_copy(adir)
178c      ⌈⌉ arch_shred(adir)
```

The Archive Create Behaviour: Left Fig. 5 on Page 40 The **archive create** behaviour

179 accepts a request, from the management behaviour to create a document;
180 it then selects an available document identifier;
181 communicates this new document identifier to the management behaviour;
182 while initiating a new document behaviour, docu_{di}, with the document descriptor, *dd*, the initial document annotation being the singleton list of the note, *an*, and the initial document contents, *dc* – all received from the management behaviour – and an initial document history of just one entry: the date of creation, all
183 in parallel with resuming the archive behaviour with updated programmable attributes.

```
178a.  arch_create: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[ di ]|di:DI•di ∈ dis} Unit
178a.  arch_create(avail,used,gone) ≡
179.   [2] let mkCreate((hi,t),dd,an,dc) = mgmt_arch_ch ? in
180.       let di:DI•di ∈ avail in
181.   [4] mgmt_arch_ch ! mkNewDocID(di) ;
182.   [3] docu_di(dd)(⟨an⟩,dc,<(date_of_creation)>)
183.       ‖ arch(avail\{di},used∪{di},gone)
178a.      end end
```

The Archive Copy Behaviour: Right Fig. 6 on Page 40 The **archive copy** behaviour

184 accepts a copy document request from the management behaviour with the identity, *j*, of the master document;
185 it communicates (the request to obtain all the attribute values of the master document, *j*) to that document behaviour;
186 whereupon it awaits their communication (i.e., (*dd*,*da*,*dc*,*dh*));
187 (meanwhile) it obtains an available document identifier,
188 which it communicates to the management behaviour,
189 while initiating a new document behaviour, docu_{di}, with the master document descriptor, *dd*, the master document annotation, and the master document contents, *dc*, and the master document history, *dh* (all received from the master document),
190 in parallel with resuming the archive behaviour with updated programmable attributes.

```
178b.  arch_copy: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[ di ]|di:DI•di ∈ dis} Unit
178b.  arch_copy(avail,used,gone) ≡
184.   [3] let mkDocID(j,hi) = mgtm_arch_ch ? in
185.       arch_docu_ch[ j ] ! mkReqAttrs() ;
186.       let mkAttrs(dd,da,dc,dh) = arch_docu_ch[ j ] ? in
187.       let di:DI • di ∈ avail in
188.       mgtm_arch_ch ! mkCopyDocID(di) ;
189.   [6,7] docu_di(augment(dd,"copy",j,hi),augment(da,"copy",hi),dc,augment(dh,("copy",date_and_time,j,hi)))
190.     ‖ arch(avail\{di},used∪{di},gone)
178b.    end end end
```

where we presently leave the [overloaded] augment functions undefined.

The Archive Shred Behaviour: Right Fig. 7 on Page 41 The **archive shred** behaviour

191 accepts a shred request from the management behaviour.
192 It communicates this request to the identified document behaviour.
193 And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

```
178c.  arch_shred: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[ di ]|di:DI•di ∈ dis} Unit
178c.  arch_shred(avail,used,gone) ≡
191.   [ 2 ]  let mkShred(j) = mgmt_arch_ch ? in
192.          arch_docu_ch[ j ] ! mkShred() ;
193.          arch(avail,used,gone∪{j})
178c.          end
```

A.17.3. Handler Behaviours

194 The handler behaviour is involved in the following action traces:

a **create** Fig. 5 on Page 40 Left
b **edit** Fig. 5 on Page 40 Right
c **read** Fig. 6 on Page 40 Left
d **copy** Fig. 6 on Page 40 Right
e **grant** Fig. 7 on Page 41 Left

```
value
194   hdlr_hi: HATTRS → in,out mgtm_hdlr_ch[ hi ],{hdlr_docu_ch[ hi,di ]|di:DI•di∈dis} Unit
194   hdlr_hi(hattrs) ≡
194a          hdlr_create_hi(hattrs)
194b        ⌈⌉ hdlr_edit_hi(hattrs)
194c        ⌈⌉ hdlr_read_hi(hattrs)
194d        ⌈⌉ hdlr_copy_hi(hattrs)
194e        ⌈⌉ hdlr_grant_hi(hattrs)
```

The Handler Create Behaviour: Left Fig. 5 on Page 40

195 The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di.
196 It according updates its programmable hattrs attribute;
197 and resumes being a handler behaviour with that update.

```
194a  hdlr_create_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[ hi ] Unit
194a  hdlr_create_hi(hattrs,hhist) ≡
195     let mkGrant(di,acrs) = mgtm_hdlr_ch[ hi ] ? in
196     let hattrs′ = hattrs † [ hi ↦ acrs ] in
197     hdlr_create_hi(hattrs′,augment(hhist,mkGrant(di,acrs))) end end
```

The Handler Edit Behaviour: Right Fig. 5 on Page 40

198 The handler behaviour, on its own volition, decides to edit a document, *di*, for which it has editing rights.
199 The handler behaviour selects a suitable (...) pair of *e*dit/*u*ndo functions and a suitable (annotation) note.
200 It then communicates the desire to edit document *di* with (*e*,*u*) (at time *t*=time_ch?).
201 Editing take some time, *ti*.

202 We can therefore assert that the time at which editing has completed is $t+ti$.
203 The handler behaviour accepts the edit completion message from the document handler.
204 The handler behaviour can therefore resume with an updated document history.

```
194b  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[ hi,di ]|di:DI•di∈dis} Unit
194b  hdlr_edit_hi(hattrs,hhist) ≡
198 [1]  let di:DI • di ∈ dom hattrs ∧ "edit" ∈ hattrs(di) in
199 [1]  let (e,u):(EDIT×UNDO) • ... , n:AN • ... in
200 [1]  hdlr_docu_ch[ hi,di ] ! mkEdit(hi,t=time_ch?,e,u,n) ;
201 [2]  let ti:TIME_INTERVAL • ... in
202 [2]  wait ti ; assert: time_ch? = t+ti
203 [3]  let mkEditComplete(ti′,...) = hdlr_docu_ch[ hi,di ] ? in assert ti′ ≅ ti
204        hdlr_hi(hattrs,augment(hhist,(di,mkEdit(hi,t,ti,e,u))))
194b    end end end end
```

The Handler Read Behaviour: Left Fig. 6 on Page 40

205 The **handler behaviour**, on its own volition, decides to read a document, di, for which it has reading rights.
206 It then communicates the desire to read document di with at time t=time_ch? – with an annotation note (n).
207 Reading take some time, ti.
208 We can therefore assert that the time at which reading has completed is $t+ti$.
209 The handler behaviour accepts the read completion message from the document handler.
210 The handler behaviour can therefore resume with an updated document history.

```
194c  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[ hi,di ]|di:DI•di∈dis} Unit
194c  hdlr_edit_hi(hattrs,hhist) ≡
205 [1]  let di:DI • di ∈ dom hattrs ∧ "read" ∈ hattrs(di), n:N • ... in
206 [1]  hdlr_docu_ch[ hi,di ] ! mkRead(hi,t=time_ch?,n) ;
207 [2]  let ti:TIME_INTERVAL • ... in
208 [2]  wait ti ; assert: time_ch? = t+ti
209 [3]  let mkReadComplete(ti,...) = hdlr_docu_ch[ hi,di ] ? in
210        hdlr_hi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
194c    end end end
```

The Handler Copy Behaviour: Right Fig. 6 on Page 40

211 The **handler [copy] behaviour**, on its own volition, decides to copy a document, di, for which it has copying rights.
212 It communicates this copy request to the management behaviour.
213 After a while the handler [copy] behaviour receives acknowledgement of a completed copying from the management behaviour.
214 The handler [copy] behaviour records the request and acknowledgement in its, thus updated whereupon the handler [copy] behaviour resumes being the handler behaviour.

```
194d  hdlr_copy_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[ hi ] Unit
194d  hdlr_copy_hi(hattrs,hhist) ≡
211   [1] let di:DI • di ∈ dom hattrs ∧ "copy" ∈ hattrs(di) in
212   [1] mgtm_hdlr_ch[ hi ] ! mkCopy(di,hi,t=time_ch?) ;
213  [10] let mkCopyComplete(di′,di) = mgtm_hdlr_ch[ hi ] ? in
214  [10] hdlr_hi(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di′))))
194d      end end
```

The Handler Grant Behaviour: Left Fig. 7 on Page 41

215 The **handler [grant] behaviour** offers to accept grant permissions from the management behaviour.
216 In response it updates its handler attribute while resuming being a handler behaviour.

```
194e  hdlr_grant_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[ hi ] Unit
194e  hdlr_grant_hi(hattrs,hhist) ≡
215   [ 2 ] let mkGrant(di,acrs) = mgtm_hdlr_ch[ hi ] ? in
216   [ 2 ] hdlr_hi(hattrs†[ di↦acrs ],augment(hhist,time_ch?,mkGrant(di,acrs)))
194e      end
```

A.17.4. Document Behaviours

217 The document behaviour is involved in the following action traces:

a **edit** Fig. 5 on Page 40 Right
b **read** Fig. 6 on Page 40 Left
c **shred** Fig. 7 on Page 41 Right

```
value
217  docu_di: DD × (DA × DC × DH) →  in,out arch_docu_ch[ di ], {hdlr_docu_ch[ hi,di ]|hi:HI•hi∈his} Unit
217  docu_di(dattrs) ≡
217a       docu_edit_di(dd)(da,dc,dh)
217b     ⌈⌉ docu_read_di(dd)(da,dc,dh)
217c     ⌈⌉ docu_shred_di(dd)(da,dc,dh)
```

The Document Edit Behaviour: Right Fig. 5 on Page 40

218 The **document [edit] behaviour** offers to accept edit requests from document handlers.

a The document contents is edited, over a time interval of ti, with respect to the handlers edit function (e),
b the document annotations are augmented with respect to the handlers note (n), and
c the document history is augmented with the fact that an edit took place, at a certain time, with a pair of *edit*/*undo* functions.

219 The *e*dit (etc.) function(s) take some time, ti, to do.
220 The handler behaviour is notified, mkEditComplete(...) of the completion of the edit, and
221 the document behaviour is then resumed with updated programmable attributes.

```
value
217a  docu_edit_di: DD × (DA × DC × DH) →  in,out {hdlr_docu_ch[ hi,di ]|hi:HI•hi∈his} Unit
217a  docu_edit_di(dd)(da,dc,dh) ≡
218   [ 2 ]  let mkEdit(hi,t,e,u,n) = []{hdlr_docu_ch[ hi,di ]?|hi:HI•hi∈his} in
218a  [ 2 ]  let dc′ = e(dc),
218b           da′ = augment(da,((hi,t),("edit",e,u),n)),
218c           dh′ = augment(dh,((hi,t),("edit",e,u))) in
219          let ti = time_ch? − t in
220          hdlr_docu_ch[ hi,di ] ! mkEditComplete(ti,...) ;
221          docu_di(dd)(da′,dc′,dh′)
217a         end end end
```

The Document Read Behaviour: Left Fig. 6 on Page 40

222 The The **document [read] behaviour** offers to receive a read request from a handler behaviour.
223 The reading takes some time to do.

224 The handler behaviour is advised on completion.
225 And the document behaviour is resumed with appropriate programmable attributes being updated.

```
value
217b  docu_read_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
217b  docu_read_di(dd)(da,dc,dh) ≡
222   [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
223   [2] let ti:TIME_INTERVAL • ... in
223   [2] wait ti ;
224   [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
225   [2] docu_di(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,"read")))
217b      end end
```

The Document Shred Behaviour: Right Fig. 7 on Page 41

226 The **document [shred] behaviour** offers to accept a document shred request from the archive behaviour –
227 whereupon it **stop**s!

```
value
217c  docu_shred_di: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
217c  docu_shred_di(dd)(da,dc,dh) ≡
226   [3] let mkShred(...) = arch_docu_ch[di] ? in
227       stop
217c  [3] end
```

A.18. Conclusion

This completes a first draft version of this document. The date time is: 10 September 2017: 05:36 am. Many things need to be done. First a careful checking of all types and functions: that all used names have been defined. The internal non-deterministic choices in formula Items 142 on Page 42, 178 on Page 44, 194 on Page 46 and 217 on the facing page, need be checked. I suspect there should, instead, be som mix of both internal and external non-deterministic choices. Then a careful motivation for all the other non-deterministic choices.

B. RSL

C. RSL: The RAISE Specification Language – A Primer

C.1. Type Expressions

Type expressions are expressions whose value are types, that is, possibly infinite sets of values (of "that" type).

C.1.1. Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

type
[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23···, 2,7182···, 3,1415···, 4.56, ...
[5] **Char** "a", "b", ..., "0", ...
[6] **Text** "abracadabra"

C.1.2. Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then the following are type expressions:

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] A $\xrightarrow[m]{}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following the meaning of the atomic and the composite type expressions:

1 The Boolean type of truth values **false** and **true**.
2 The integer type on integers ..., –2, –1, 0, 1, 2, ...
3 The natural number type of positive integer values 0, 1, 2, ...
4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5 The character type of character values ″a″, ″bb″, ...
6 The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7 The set type of finite cardinality set values.
8 The set type of infinite and finite cardinality set values.
9 The Cartesian type of Cartesian values.

10 The list type of finite length list values.
11 The list type of infinite and finite length list values.
12 The map type of finite definition set map values.
13 The function type of total function values.
14 The function type of partial function values.
15 In (A) A is constrained to be:
   - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
   - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.
16 The postulated disjoint union of types A, B, ..., and C.
17 The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18 The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

```
type
  A, B, C, ..., D
value
  obs_B: A → B, obs_C: A → C, ..., obs_D: A → D
```

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

```
type
  B, C, ..., D
  A = B × C × ... × D
```

C.2. Type Definitions

C.2.1. Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

```
type
  A = Type_expr
```

Some schematic type definitions are:

```
[19] Type_name = Type_expr /∗ without |s or subtypes ∗/
[20] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[21] Type_name ==
                mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
                ... |
                mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[22] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[23] Type_name = {| v:Type_name′ • P(v) |}
```

where a form of [20]–[21] is provided by combining the types:

```
  Type_name = A | B | ... | Z
  A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
  B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
```

...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom
∀ a1:A_1, a2:A_2, ..., ai:Ai •
  s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
  ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
∀ a:A • **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
  a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) **end**

C.2.2. Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

type
A = {| b:B • $\mathcal{P}$(b) |}

C.2.3. Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

type
A, B, ..., C

C.3. The RSL Predicate Calculus

C.4. Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

C.4.1. Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

C.4.2. Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

C.5. Concrete RSL Types: Values and Operations

C.5.1. Arithmetic

type
  Nat, **Int**, **Real**
value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\xrightarrow{\sim}$**Nat** | **Int**×**Int**$\xrightarrow{\sim}$**Int** | **Real**×**Real**$\xrightarrow{\sim}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

C.5.2. Set Expressions

Set Enumerations Let the below a's denote values of type A, then the below designate simple set enumerations:

{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ...} ∈ A-**set**
{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ..., {e$_1$,e$_2$,...}} ∈ A-**infset**

Set Comprehension The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

C.5.3. Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

C.5.4. List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:

```
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension The last line below expresses list comprehension.

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: Aω × P × Q ~→ Bω
  comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

C.5.5. Map Expressions

Map Enumerations Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

```
type
  T1, T2
  M = T1 m→ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] all ∈ M
```

Map Comprehension The last line below expresses map comprehension:

```
type
  U, V, X, Y
  M = U m→ V
  F = U ~→ X
  G = V ~→ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X m→ Y)
  comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

C.5.6. Set Operations

Set Operator Signatures

```
value
  19 ∈: A × A-infset → Bool
  20 ∉: A × A-infset → Bool
  21 ∪: A-infset × A-infset → A-infset
  22 ∪: (A-infset)-infset → A-infset
  23 ∩: A-infset × A-infset → A-infset
  24 ∩: (A-infset)-infset → A-infset
  25 \: A-infset × A-infset → A-infset
  26 ⊂: A-infset × A-infset → Bool
  27 ⊆: A-infset × A-infset → Bool
```

28 =: A-infset × A-infset → Bool
29 ≠: A-infset × A-infset → Bool
30 card: A-infset $\xrightarrow{\sim}$ Nat

Set Examples

```
examples
  a ∈ {a,b,c}
  a ∉ {}, a ∉ {b,c}
  {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪{{a},{a,bb},{a,d}} = {a,b,d}
  {a,b,c} ∩ {c,d,e} = {c}
  ∩{{a},{a,bb},{a,d}} = {a}
  {a,b,c} \ {c,d} = {a,bb}
  {a,bb} ⊂ {a,b,c}
  {a,b,c} ⊆ {a,b,c}
  {a,b,c} = {a,b,c}
  {a,b,c} ≠ {a,bb}
  card {} = 0, card {a,b,c} = 3
```

Informal Explication

19 ∈: The membership operator expresses that an element is a member of a set.
20 ∉: The nonmembership operator expresses that an element is not a member of a set.
21 ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22 ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23 ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24 ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
25 \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26 ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27 ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28 =: The equal operator expresses that the two operand sets are identical.
29 ≠: The nonequal operator expresses that the two operand sets are *not* identical.
30 **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (≡ is the definition symbol):

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
```

```
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

C.5.7. Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

C.5.8. List Operations

List Operator Signatures

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⌢: A* × A^ω → A^ω
```

List Operation Examples

```
examples
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
  ⟨a1,a2,...,am⟩(i)=ai
  ⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
  ⟨a,b,c⟩=⟨a,b,c⟩
  ⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.

- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }


  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩^q′ in a end
          else chaos end
      else q(i−1) end

  fq ^ iq ≡
        ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
        | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

C.5.9. Map Operations

Map Operator Signatures and Map Operation Examples

```
value
  m(a): M → A ~→ B, m(a) = b

  dom: M → A-infset [domain of map]
    dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

  rng: M → B-infset [range of map]
    rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

  †: M × M → M [override extension]
    [a↦b,a′↦bb′,a″↦bb″] † [a′↦bb″,a″↦bb′] = [a↦b,a′↦bb″,a″↦bb′]
```

∪: M × M → M [merge ∪]
  [a↦b,a′↦bb′,a″↦bb″] ∪ [a‴↦bb‴] = [a↦b,a′↦bb′,a″↦bb″,a‴↦bb‴]

\: M × A-**infset** → M [restriction by]
  [a↦b,a′↦bb′,a″↦bb″]\{a} = [a′↦bb′,a″↦bb″]

/: M × A-**infset** → M [restriction to]
  [a↦b,a′↦bb′,a″↦bb″]/{a′,a″} = [a′↦bb′,a″↦bb″]

=,≠: M × M → **Bool**

°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
  [a↦b,a′↦bb′] ° [bb↦c,bb′↦c′,bb″↦c″] = [a↦c,a′↦c′]

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation Redefinitions

The map operations can also be defined as follows:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

C.6. λ-Calculus + Functions

C.6.1. The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

C.6.2. Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

C.6.3. Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

C.6.4. α-Renaming and β-Reduction

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

C.6.5. Function Signatures

For sorts we may want to postulate some functions:

```
type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: BB×C → A
```

C.6.6. Function Definitions

Functions can be defined explicitly:

```
value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments ~→ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```

Or functions can be defined implicitly:

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ~→ Result
  g(args) as result
  pre P2(args)
  post P3(args,result)
```

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

C.7. Other Applicative Expressions

C.7.1. Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

C.7.2. Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **YF** **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

C.7.3. Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

C.7.4. Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,bb⟩^ℓ = list **in** ... **end**

let [a↦bb] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

C.7.5. Conditionals

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
    if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
    choice_pattern_1 → expr_1,
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n
end
```

C.7.6. Operator/Operand Expressions

⟨Expr⟩ ::=

⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...
⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !

C.8. Imperative Constructs

C.8.1. Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Unit
value
stmt: **Unit** → **Unit**
stmt()

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

C.8.2. Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

C.8.3. Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

C.8.4. Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

C.8.5. Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

C.8.6. Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

C.9. Process Constructs

C.9.1. Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

C.9.2. Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

P ‖ Q Parallel composition
P [] Q Nondeterministic external choice (either/or)
P ⨅ Q Nondeterministic internal choice (either/or)
P ╫ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal (⨅). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

C.9.3. Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

C.9.4. Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

value
P: **Unit** → **in** c **out** k[i]
Unit
Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

C.10. Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type
  ...
variable
  ...
channel
  ...
value
  ...
axiom
  ...

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[64]

[64] For schemes, classes and objects we refer to [9, Chap. 10]